Modelling Parallel Database Management Systems for Performance Prediction

Neven T. Tomov

Thesis submitted for the degree of

Doctor of Philosophy

Department of Computing and Electrical Engineering

Heriot-Watt University

Edinburgh

June 1999

DECLARATION

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, except where due acknowledgement is made, and has not been submitted for any other degree.

Neven T. Tomov

June 1999

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I express my gratitude to all those who made this thesis possible.

Howard Williams was a constant source of intellectual support, without which this thesis would have been a very difficult task. Albert Burger, Lachlan MacKinnon and Hamish Taylor took time to read my thesis carefully; their insights and constructive criticism helped me to improve it considerably. Peter King and Phil Trinder were a source of useful discussions and I thank them for their helpful and incisive comments.

I am especially indebted to my colleague Euan Dempster for the many hours spent discussing and improving my ideas, and to Shaoyu Zhou who helped me get started in research. Both have contributed significantly to the ideas in this thesis.

I thank ICL for providing access to the Goldrush machines. I express my gratefulness to Phil Broughton, Arthur Fitzjohn and Monique Mitchel who helped my understanding of the realities of parallel databases.

I also wish to thank the Commission of the European Union and the UK Engineering and Physical Sciences Research Council (EPSRC) for providing the research grants that supported my work.

The following friends and colleagues were a source of invaluable discussions and diversions, especially towards the end: Adam Cobham, John Grady, Chai-Seng Pua, Phil Spencer, Nikos Vassilopoulos, and Caitriana White.

I dedicate this work to my parents, Toma and Ivanka, who provided me with the inspiration to pursue a PhD. I owe a debt to my wife Alexandra for putting up with my long hours and for the support, love and encouragement, without which this work would not have been possible.

ABSTRACT

This thesis proposes an analytical approach to modelling parallel relational database systems to produce estimates of average transaction response time, resource utilisation and maximum throughput. The modelling approach is applicable to commercial systems, such as Informix and Oracle, and can serve as a performance prediction core of tools that can aid in application design, system sizing and capacity planning for parallel databases.

A representation of parallel relational database activity is developed. Applications are mapped to a low-level representation of patterns of resource consumption, capturing the execution logic of relational operators, and mechanisms such as pipelined and partitioned execution. From this, resource utilisation and maximum throughput are derived.

A novel method for transaction response time estimation is proposed. Resource usage profiles are mapped to open multi-class queueing networks. Firstly, queue waiting times are estimated given the involvement of resources in database activity. Since the networks are not in product form, a heuristic rule is developed to approximate queue waiting times. The rule's effectiveness when compared with more complex techniques is demonstrated. Unlike other approximation techniques, it performs consistently well over a wide range of examples. Secondly, the average response time of a transaction is obtained from the estimated waiting times and the original resource usage profile. Synchronisation mechanisms such as pipelines between operators and partitioned parallelism are taken into account.

The modelling approach is implemented within a performance estimation tool. To validate the approach comparisons of estimated vs. measured Informix performance are presented.

Effects of background database activity, e.g. cache maintenance, also need to be taken into account when predicting performance. Cache models are developed for Oracle and Informix.

Chapter 1

INTRODUCTION

1.1 Background

Exploiting parallelism in database systems has proved very successful and has been the key to building high-performance database systems. Parallel database systems are recognised as one of the more successful applications of parallel computer technology [34]. In recent years several prototype systems have been developed within research projects [13-17, 22-26]. Also, several modern commercial database systems have been adapted to produce parallel versions [6-12, 27-28]. Various architectures (e.g. shared-nothing, shared-disc) and techniques (e.g. relational pipelines, inter-operator and intra-operator parallelism, parallel algorithms for join, data placement, load balancing, cache coherency control and so on) have been studied and are used to harness parallelism coherently and efficiently [18].

The ability to predict or estimate the performance of computer systems has always been important [55, 103, 63]. This is especially true in the case of parallel database systems. It can be used before acquiring a parallel system to determine a suitable configuration, or once one has a parallel database system to tune its performance or decide how best to upgrade it. For example, there are many alternative ways of breaking up and distributing the data across the nodes and discs of a system, or scheduling an operator tree for execution. Performance prediction tools that automate the process of comparing and evaluating design decisions would benefit application designers, system sizers and database administrators.

However, due to the complexity of parallel databases, performance estimation for such systems is a difficult task. Correspondingly, the design of performance prediction tools and methodologies for this is a challenging task.

1.2 Summary of thesis

This thesis proposes an approach to modelling parallel database systems so that performance measures such as average transaction response time and maximum system throughput may be estimated. The aim has been to develop a practical approach, applicable to real systems. In particular, the models are informed by the ICL Goldrush platform and the Informix and Oracle parallel engines. This provides an obvious measure of success: the results produced by the model should compare well against measurements obtained from the modelled systems.

The approach consists of three stages. A preliminary stage establishes a representation of parallel database activity, suitable for the performance estimation task. The representation is a transformation of a database application to a description of patterns of resource consumption by the components of the system. This *resource usage profile* captures the execution 'logic' of relational operators, such as hash-join, as well as types of parallelism such as relational pipelines and partitioned parallelism. Resource utilisation and maximum system throughput can be derived directly from the resource usage profile.

The second and third stages of the approach are concerned with response time estimation of transactions. Together with maximum throughput this is an important indicator of performance, but is more difficult to estimate. Resource usage profiles are mapped to open, multi-class queueing networks. Such networks can be 'solved' and response time obtained. Thus, contention for resources, which inevitably occurs when a number of processes or threads compete for service, is taken into account in estimating the response time. Unfortunately, 'solving' the obtained networks is not straightforward

since they are not in product form, due to both non-exponential service times at queueing centres and synchronisation between query execution phases (pipelines, partitioned parallelism), which imposes synchronisation on the transitions between queues. For such networks well-known exact analytical solutions do not apply.

A two-step solution is proposed to tackle this problem. First, leaving synchronisation aside, obtain queue waiting time estimates for individual resources. Second, using these estimates, form an estimate of transactions response time, taking into account synchronisation and dependencies between query execution phases.

The second stage of the approach is therefore concerned with estimating the waiting times at queueing network centres, given their involvement in database activity as specified by the resource usage profile. The difficulty here comes from the non-exponential service times of queueing centres, which renders exact analytical solutions unusable. A number of approximation techniques developed in the literature, some of which are quite complex, can be used to estimate queue waiting time. To choose the most appropriate one, a study is conducted in which a selection of techniques are applied to a large number of examples. From the results a heuristic rule is derived.

Finally, the third stage of the approach estimates the average transaction response time. Given the estimated resource waiting times and the original resource usage profile, this is obtained through a traversal of the profile, paying careful attention to the types of synchronisation that govern the execution of a query in a parallel environment. Examples of such mechanisms include relational pipelines between two or more operators of the execution schedule, partitioned parallelism within a single operator, and blocking.

Other types of database behaviour (apart from query execution) can be taken into account within the framework of the method. To show this, models of the cache management mechanisms of Oracle and Informix are developed to predict cache hit

probabilities that can be incorporated into the resource usage profile and thus taken into account in further performance estimation stages.

The proposed performance estimation approach is capable of producing estimates rapidly. To illustrate its utility, it has been implemented as part of a stand-alone tool for rapid performance prediction. Some comparisons are presented of performance estimated by the tool against actual performance. These illustrate the validity and applicability of the approach.

1.3 Contribution of thesis

The work reported in this thesis was carried out within two collaborative projects at the Department of Computing and Electrical Engineering at Heriot-Watt University: the MERCURY project under the Framework IV programme of the Commission of the European Union and the Tools project under PSTPA programme of the UK Engineering and Physical Sciences Research Council (EPSRC). The broad objective of the two projects was to promote the wider use of parallel computer technology. In particular, the projects' goal was the development of tools to aid with capacity planning, application sizing and data placement in parallel relational database systems.

The thesis describes the approach to performance modelling developed within the projects, which serves as the core of a tool for rapid performance prediction in parallel relational databases. The focus of the thesis and its main contributions are as follows:

1. A novel analytical approach to response time estimation is developed. The approach is based on a representation of database activity during query execution in terms of patterns of resource consumption, and consists of:
    - a novel heuristic rule for solving open queueing networks with non-exponential service times;

- formulae for estimating the response time of queries, which take into account pipelined execution and intra-operator parallelism.

2. The response time technique is validated for Informix XPS on the ICL Goldrush system. A selection of queries of different complexity are executed and the performance of the system is measured and compared against the predicted values. The results show that the method produces acceptable results, which are typically within 20% of the measured values.
3. Cache models for Oracle Parallel Server and Informix XPS are developed, based on earlier work by Dan and Yu [1, 2]. This is done to illustrate how background database activity, in addition to query execution, may be taken into account within the modelling approach.
4. A performance prediction tool, which implements the response time and cache models, is developed. The tool extends the capabilities of the previously developed STEADY [3, 4] tool.

As explained above, the work reported here was carried out within a collaborative project, involving three other researchers. The development of the representation of database activity (Chapter 4), the calibration and validation runs (Chapter 7) and the implementation of the tool itself was the result of joint work with colleagues.

1.4 Structure of thesis

Chapter 2 is an introduction to parallel database systems. Three common architectural models are described and relevant terminology is introduced. Benchmarks are introduced. Three commercial systems are described: the Goldrush hardware platform from ICL and two parallel relational database systems - Informix XPS and Oracle Parallel Server - designed to take advantage of its parallel architecture. All three have been used in the development of the performance models developed in the thesis.

Chapter 3 focuses on performance estimation and modelling in the context of parallel database systems. General approaches to performance modelling are introduced. Some examples of performance estimation studies in parallel database systems and tools are given. The chapter concludes with an outline of STEADY, a tool that provides the starting point of the work carried out in the thesis.

Chapter 4 develops a representation of database activity. Examples are used to illustrate the process of obtaining the resource usage profile of a given query. A description is provided of how resource utilisation and maximum system throughput is obtained from the representation.

Chapters 5 and 6 are devoted to response time estimation. Chapter 5 proposes a method for estimating the queue waiting time of physical resources, given their involvement in database activity as specified in the resource usage profile. Chapter 6 discusses how, given a resource usage profile and approximated queue waiting times, an overall response time for a transaction can be obtained.

Chapter 7 provides a validation of the response time estimation technique developed in Chapters 4, 5 and 6. A number of tables and queries are considered. The response times of the queries, obtained from measurements performed on an Informix XPS Goldrush system, are compared against the estimations produced by STEADY.

Chapter 8 develops cache models for Oracle Parallel Server and Informix XPS on Goldrush. The models are intended to capture the way the two systems maintain and administer database buffers in memory, and serve to illustrate how other type of database activity may be accounted for when estimating performance.

Chapter 9 concludes the thesis and suggests directions for possible future work.

1.5 Publications

The following papers have been published:

1) Zhou S, Tomov N, Williams M.H, Burger A and Taylor H, "Cache Modelling in a Performance Evaluator of Parallel Database Systems", *Proc. of the 5th Int. Symp. on*

Modelling, Analysis and Simulation of Computer and Telecommunication Systems (*MASCOTS '97*), Haifa, Israel, IEEE Computer Society Press, pp. 46-50, January 1997;

2) E. W. Dempster, N. T. Tomov, J. Lu, C. S. Pua, M. H. Williams, A. Burger, H. Taylor and P. Broughton, "Verifying a Performance Estimator for Parallel DBMSs", *Proc. of the 4th Int. Euro-Par Conf.* (*Euro-Par '98 Parallel Processing*), Southampton, UK, Lecture Notes in Computer Science No. 1470, Springer-Verlag, pp. 126-135, September 1-4, 1998;

3) N. Tomov, E. Dempster, M. H. Williams, P. Broughton, "Some Results from a New Technique for Response Time Estimation in Parallel DBMS", *Proc. of the 7th Int. Conf. and Exhibition on High-Performance Computing and Networking* (*HPCN Europe '99*), Amsterdam, The Netherlands, Lecture Notes in Computer Science No. 1593, Springer-Verlag, pp. 713-721, April 12-14 1999.

The following have been accepted for publication and will appear in due course:

4) M. H. Williams, E. Dempster, N. Tomov, C. Pua, H. Taylor, A. Burger, J. Lu, P. Broughton, "An analytical tool for predicting the performance of parallel relational databases", *Concurrency: Practice and Experience*, John Wiley & Sons Ltd., to appear, 1999;

5) N. Tomov, E. Dempster, M. H. Williams, P. King, A. Burger, "Approximate estimation of transaction response time ", *The Computer Journal*, Oxford University Press, to appear, 1999.

The following paper has been submitted for publication:

6) N. Tomov, E. Dempster, M. H. Williams, A. Burger, H. Taylor, P. King, P. Broughton, "Practical response time estimation in parallel relational database systems", submitted to *IEEE Trans. Parallel and Distributed Systems*, July 1999.

Chapter 2

PARALLEL RELATIONAL DATABASE SYSTEMS

2.1 Introduction

The purpose of this chapter is to provide a broad overview of the field of parallel relational database systems, introducing concepts and identifying current research areas. In this way the research reported in this work can be seen in a proper context.

Parallel relational database systems have emerged as one of the most successful application areas of conventional parallel computer technology. They provide an answer to increasing user demand for higher performance. Prior to the emergence of parallel relational database systems for conventional multiprocessors, there had been a lot of research into so-called parallel database machines – specialised software running on purpose-built hardware, well suited to doing relational-type operations in parallel [5]. Such specialised machines have been displaced by database solutions for standard SMP (Symmetric Multiprocessor) and MPP (Massively Parallel Processor) architectures, which have become both widespread and affordable.

Existing parallel relational database systems are usually classified as shared-everything, shared-disc and shared-nothing, although the distinctions are becoming blurred. Section 2.2 provides a description of the three types of systems in broad terms, and discusses their relative advantages and shortcomings. The introduction of parallelism to relational systems has led to new concepts and problems, specific to each architecture type. This has focused research effort into various areas. Some relevant issues and corresponding research are outlined in Section 2.3. Some important benchmarks, used to evaluate systems and designs, are introduced in Section 2.4.

Many of the ideas from research have been taken up by industry, and today, major hardware and software vendors provide parallel database solutions (e.g. Informix [6], Oracle [7], DB2 from IBM [8], Teradata [9], Compaq NonStop SQL [10], Sybase [11], SQL Server from Microsoft [12]). Section 2.5 focuses on three commercial products, the Goldrush MegaServer parallel platform from ICL and two relational database systems implemented on it – the Informix Extended Parallel Server and the Oracle Parallel Server with Parallel Query Option.

2.2 Review of parallel database architectures

The majority of parallel architectures which host database systems comprise three main types of components: processors, main memory modules, and secondary storage (discs). The topology in which various components are connected reflects architectural design choices made by hardware vendors and research teams as to how multiple components are incorporated into a single computer. It also serves as the basis for a classification of parallel database systems into three categories: shared memory, shared disc, and shared nothing.

2.2.1 Shared memory systems

In *shared memory* (or *shared everything)* database systems the discs and the memory are shared among all processors. Such systems are most naturally implemented for SMP (Symmetric Multiprocessor) machines. A typical SMP architecture consists of processors sharing the memory and disc modules via a shared bus. A generic shared memory system is represented schematically in Figure 2-1.

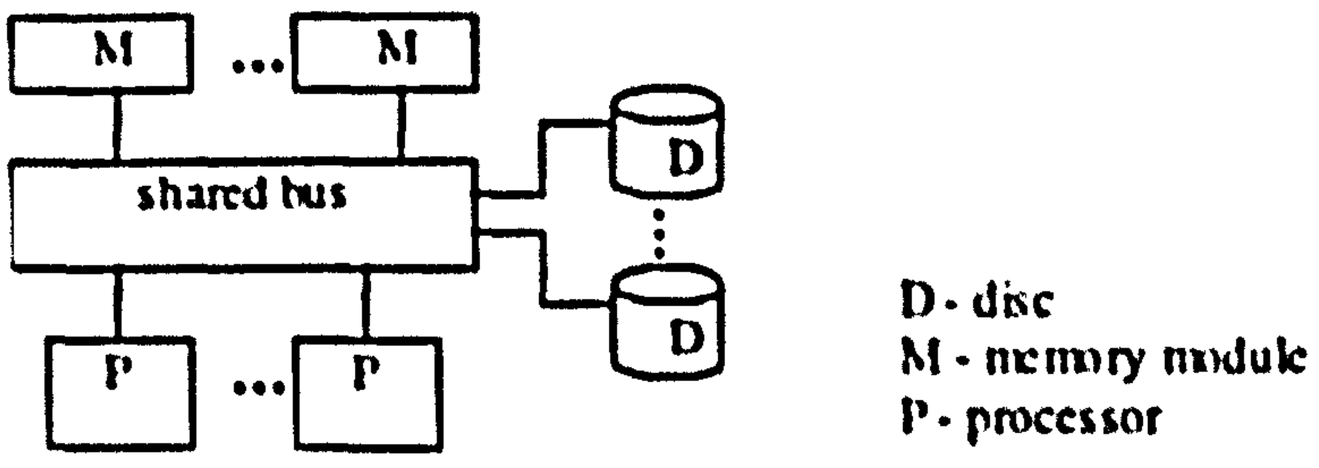


Figure 2-1 Shared memory system

Examples of research systems, developed for the shared memory model include DBS3 [13, 14], Volcano [15, 16], and XPRS [17], while examples of commercial systems include Sybase System 11 [11], Oracle7 [7], Informix OnLine [6], and SQL Server [12].

The shared bus, which allows a memory module to be accessed by any processor, increases the memory-access latency. For this reason, processors usually have fast private cache memories for speeding up the access to shared memory (by 5 to 50 times) [18]. However, mechanisms are required to ensure cache coherency. An example of a cache coherency algorithm is the "snooping bus write invalidate" method [19]. All cache controllers listen on the shared bus and when a memory write on a cached address occurs, the cached value is invalidated. When a read occurs, the value is read into the cache from memory. Cache coherency introduces additional overhead, which increases as the number of processors or memory modules increases. This overhead limits the potential scaleability of SMP systems and configurations with hundreds of processors are unlikely to be practical. One of the largest SMP systems available today is the Sun Enterprise 1000 Server [20], which can support up to 64 processors.

A shared memory system runs a single copy of the operating system and a single instance of the database engine. Since each processor has access to all available memory, communication is straightforward: message exchange and data sharing are through shared memory. Synchronisation is also relatively easy to implement by low-level mechanisms (such as semaphores). In addition, shared memory systems lend themselves to load balancing, usually taken care of by the operating system. SMP systems perform best when running a large number of small tasks such as multiple threads. Correspondingly, modern database systems are developed with a multithreaded server core.

2.2.2 Shared nothing systems

The term *shared nothing* [21] refers to database architectures in which every processor has its own memory and discs. Shared nothing systems are usually associated with MPP (massively parallel processing) architectures. A typical MPP system consists of a number of nodes connected via very fast point-to-point communication hardware (an interconnecting network). Each node of such a MPP system consists of a processor, memory module(s), and discs. Nodes run separate instances of the operating system as well as instances of the database engine. The shared nothing architecture is given schematically in Figure 2-2.

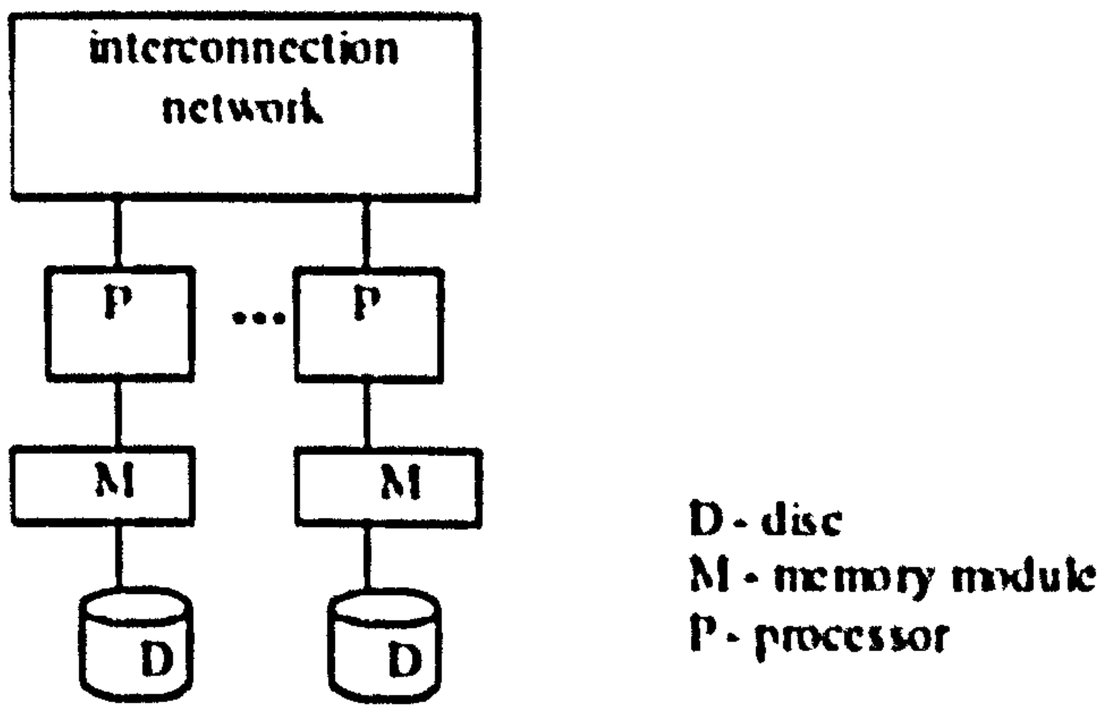


Figure 2-2 Shared nothing system

Examples of research systems, developed for the shared nothing model are Gamma [22, 23], Bubba [24, 25] and EDS [26], while commercial products include Informix XPS [27], Sybase MPP [11], DB2 Parallel Edition [8] and Compaq NonStop SQL [10].

Each node is granted exclusive access to a portion of the database. A node has its own set of local discs where the node's data is stored. No direct access is possible from one processor to the discs and memory of another. Communication between processors is achieved by sending messages through the interconnection network. The database is partitioned and stored on the discs of different nodes.

In more recent MPP architectures, each node itself may be an SMP system. Such MPP architectures are a combination of shared memory at the node level and

distributed memory at the global system level. Moreover, the disc controllers of a particular node may also be connected to the bus of a second node in order to allow fault-tolerance. Thus the term 'shared nothing', originally referring to architectures where a disc connects to a single node with a single processor, is also used to describe hybrid systems with some limited form of memory and disc sharing.

As more nodes are added to the system, the aggregate bandwidth requirement scales up in proportion to the number of processors added. Most current interconnection networks are designed to satisfy this requirement. Thus the major advantage of MPP systems is that they can scale up to handle hundreds of nodes.

On the other hand it is difficult to co-ordinate such a large number of nodes. This difficulty manifests itself in load balancing. For a well performing system, the workload should ideally be spread evenly among collaborating nodes. In the presence of data skew (see Section 2.3.3), however, a parallelised operation such as join may not bring good returns, unless load-balancing issues are considered explicitly as part of the design of the join algorithm.

2.2.3 Shared disc systems

Shared disc refers to database systems developed for architectures which allow a number of separate nodes (comprising a processor and memory module(s)) shared access to a number of discs. Nodes are connected by an interconnecting mechanism, which allows each processor to access directly any of the discs of the system. This is represented in Figure 2-3.

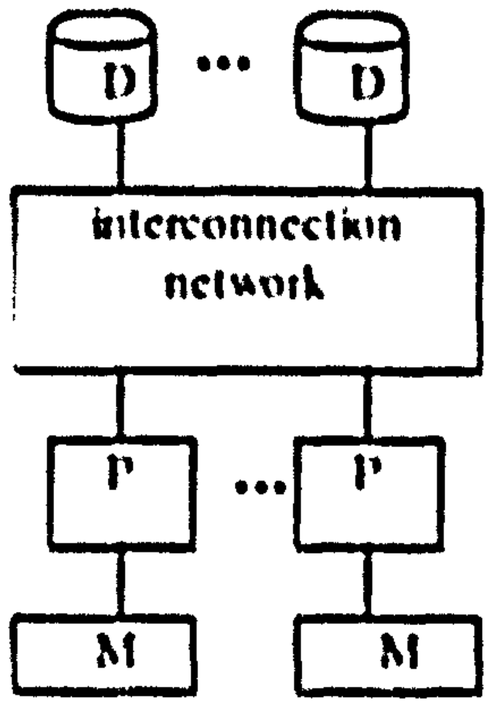


Figure 2-3 Shared disc system

Similar to the shared nothing approach each processor manages its own memory and cannot access data in the memory of other processors. Nodes communicate by sending messages across the interconnecting mechanism. Data can also be communicated directly between nodes, or the shared disc subsystem may be used for this purpose. Each node in a shared disc system may be an SMP system in its own right. Examples of commercial shared disc products include Oracle Parallel Server [7] and DB2 for IBM System/390 [28].

Shared disc systems were originally used in cluster architectures. Clusters were developed to address the scaleability limitations of shared memory systems as well as the need for increased resilience to failure. They combine two or more separate computers (typically SMP) into a single system. The interconnection mechanism (or coupler) used to connect the nodes and the disc subsystem is of high speed, but not as sophisticated as the interconnecting network of MPP architectures. Clusters are primarily used to add a second SMP system to an existing one and thus add fault-tolerance.

Shared disc systems can also be established on MPP architectures, if the disc controllers and input/output controllers are connected to the interconnecting network instead of directly to nodes via their bus. Alternatively, some vendors implement a single global file store for their MPP systems thus providing a virtual shared disc architecture. Even though certain discs in the system may be directly connected to

certain nodes, the DBMS has no concept of a node 'owning' a certain set of discs. Rather, all discs and data are accessible by all nodes.

To avoid conflicting accesses to the same pages, global locking and protocols for the maintenance of cache coherency are needed [29]. A distributed lock manager is responsible for ensuring the consistency of data. This has the potential of creating heavy traffic on the interconnection network consisting of lock protocol messages. This is not an issue with shared nothing systems, where nodes do not have to co-ordinate data sharing since each one is responsible for its own data. However, shared disc systems are considered more flexible in the sense that any node may be selected to access any piece of data, while in shared nothing systems data can be accessed only through the node that owns it.

2.2.4 Comparative analysis

A study of the benefits of the different architectures ([29]) points to extensibility and availability as the main virtues of shared nothing systems. On the other hand, the high complexity and poor load balancing are shared nothing's main disadvantages. The difficulty with load balancing is attributed to the fact that the amount and location of processing is determined by the database partitioning. This is especially a problem when a new processor or disc is to be added: a decision must be made as to how to redistribute the data among the processors and discs, which is a costly operation. The high complexity of shared nothing systems refers to the need for distributed database techniques, such as 2-phase commit, to be implemented to ensure correct functioning of the system.

The simplicity and ease of load balancing are given as shared memory's main advantages, while limited extensibility and low availability are this architecture's disadvantages. Shared memory architectures suffer from poor availability, in the sense that a memory fault may affect most processors and bring the system down.

Shared disc architectures are said to have more or less similar pros and cons to shared nothing systems, but are seen to provide a better opportunity for load balancing and ease of migration. Migrating from a centralised system to a shared disc one is relatively straightforward since data on disc does not need to be reorganised. The access of the shared disc subsystem is pointed out as a potential bottleneck. On the other hand, the shared disc ensures that a failed node will not affect the availability of the system.

Another approach to identifying the relative benefits of the shared memory, shared nothing and shared disc approaches is to use simulation models of the three architectures in order to compare their performance. Examples of this are given in Section 3.3.1, which discusses performance modelling in parallel database systems.

There is no single database architecture that is better than others in all respects. The choice of optimal platform ultimately depends on the specific user application and workload. Although a generalisation, it can be said that database solutions for SMP systems are best suited for OLTP (On-line Transaction Processing) applications, while MPP based systems are best suited for complex analytical or very large decision support applications.

The architectural differences between the three types of systems are becoming more and more blurred. Non-uniform memory access (NUMA) systems are being developed by vendors such as Data General, IBM, ICL, NCR, among others [30, 31]. In contrast to the MPP model, where each SMP node has a private memory, in NUMA systems an SMP node has access to the memory of a different node. Thus a common global memory is established. Each processor accesses the memory over a two level interconnecting mechanism. Access to local memory (memory residing on the same board as the processors of the SMP node) is through a high-speed local bus and has a low access time. Access to remote memory is through a different system bus and has higher access time. Like conventional SMP systems, a NUMA system runs a single

instance of the operating system, thus (potentially) providing a shared memory programming model with the scalability of MPP systems. The effectiveness of this architecture depends on maintaining a high level of data locality in each node.

2.3 Issues in parallel database systems

This section introduces some issues and terminology pertaining to parallel database systems that will be used throughout the rest of the thesis. Parallel query execution, data partitioning and placement, hash-based join, and query optimisation and scheduling are discussed. The issue of cache coherency is discussed in Chapter 8.

2.3.1 Parallel query execution

The relational model [32] provides ample opportunity for exploiting parallelism, because it is based on operations performed on data structures, which are uniform sets of tuples. Different forms of parallelism can be employed in query execution in order to exploit this opportunity.

Within a relational database engine, a query is represented and executed as an *operator tree*, which is derived from the query execution plan. An operator (node of the tree) represents an atomic action (execution phase) or piece of code into which the relational operations from the execution plan are decomposed. Examples of operators are *scan* (part of select operation), *build* and *probe* (parts of a join operation implemented using a hash-join algorithm), and *formRuns*, *mergeRuns* and *merge* (parts of sort-merge-based join operation). Each operator works on one or more streams of tuples and produces a new stream. Hence, operators can be arranged in parallel dataflow graphs.

With dataflow graphs of this type two types of parallelism can be identified. With *inter-query parallelism* multiple server processes are used simultaneously to work on the operator trees of different queries. In addition, *intra-query parallelism* can also

be exploited, in which multiple server processes or threads can be scheduled to execute the operators within a single query.

Intra-query parallelism itself has three variants. If neither of two operators use data produced by the other, both can run in parallel on separate processors. This is termed *independent* parallelism.

Inter-operator or *pipeline parallelism* refers to the running of a number of operators in a pipeline, with the output stream of tuples from one operator forming the input of the next one of the pipeline. Pipelines are typically implemented using a flow control mechanism, such as table queues [33] so that a fixed amount of memory is dedicated to the pipeline. With flow control, fast producers are slowed down by slow consumers (or vice-versa), thereby ensuring that producer and consumer operators run concurrently. As [34] points out, operator pipelines of more than two operators can be constructed, but pipelines of length 10 or more are unusual. One reason for this is that some operators do not emit their first output until they have consumed the whole of their input. For example, operators such as *aggregate* (*sum*, *count*, etc.) and *sort* can not be pipelined.

The third form of intra-query parallelism is *intra-operator* or *partitioned parallelism*. With this form of parallelism several processes are assigned to work together on a single operator of the tree, such as *scan* or *aggregate*. This partitioned execution can be achieved either by designing and implementing parallel algorithms for each operator, or by parallelising the data instead [5]. This is simpler, since the use of existing routines for executing operators is preserved, and is achieved by inserting *merge* and *split* operators in suitable places within the dataflow graph.

The *merge* operator combines several data streams into a single sequential stream to be fed into the next operator. In this way, an operator working on a stream of tuples can be executed as a number of sub-processes executing on independent

processors, each working on a subset of the tuples from the stream. The output of each operator is fed into a merge operator, where the streams are combined and passed on.

The *split* operator does the opposite: it partitions or replicates a single data stream into several independent streams, so that multiple processes may operate on the stream in parallel. The partition of the data can be round robin or hash, or can be performed by an arbitrary program [35, 36].

2.3.2 Data partitioning and placement

As discussed previously, the data of a parallel relational database system resides on a subset of the discs of the system. Typically the tuples of a single relation are *partitioned* into fragments, which are *placed* on nodes and discs so that the database system can exploit the I/O bandwidth of multiple discs by reading in parallel. The partitioning and placement of the data is closely linked to intra-operator parallelism since it determines the first step in partitioned execution (the scanning of the base relations). Intra-operator parallelism in stages further up the tree can achieve arbitrary partitioning or redistribution of data with the help of the split and merge operators.

Three main types of partitioning (also called declustering) are used [34, 5]. In *round robin partitioning*, tuples are assigned in a round robin fashion to a number of fragments. In *hash partitioning* a hash function is applied to an attribute of each tuple in order to determine which fragment the tuple belongs to. Hash partitioning allows exact match selection operations on the partitioning attribute to be directed to a single fragment so that involvement of other fragments can be avoided. In *range partitioning* contiguous attribute ranges of an attribute are mapped to different fragments. Informix XPS (see Section 2.5.2) has particularly good partitioning capabilities: it allows almost arbitrary partitioning of a table on the basis of the WHERE clause syntax of SQL.

The issue of placing the relation fragments obtained from declustering onto the nodes and discs of the parallel system is referred to as *data placement*. While fragments

may be placed on any of the dises of shared memory and shared dise architectures and be accessible from any node, data placement for shared nothing architectures is of substantial importance to performance. Some placement schemes aim to minimise the communication cost of moving data around [3]. Others aim to balance processing load across the system [3, 37]. Such load balancing schemes attempt to place fragments on the nodes to obtain an even distribution of either the amount of data or overall access frequency across the nodes.

No single data placement strategy is better than others in all respects. A comprehensive analysis of data placement issues is provided in [37] and [38].

2.3.3 Join in parallel databases

Join is one of the most frequent and costly relational operations and a lot of research has gone into the development of join algorithms for parallel environments.

Conceptually, the join of two relations A and B needs to compare every tuple from A with every tuple from B and check whether they have the same join attribute values. The result of the join consists of new tuples that are concatenations of tuples from A and tuples from B such that $A.x = B.y$, where $A.x$ and $B.y$ are the join attributes. A large number of join algorithms have been developed, and can be classified in three categories: nested-loops, sort-merge and hash-based [18].

The hash-based join is particularly suited to parallelisation. The basic hash-based join method partitions two relations into n disjoint pairs of hash buckets $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$, in such a way that for any two tuples a and b where $a \in A_i$ and $b \in B_j$ if $i \neq j$ then $A.a \neq B.b$. In this way A_i needs to be joined only with B_i. Thus (using $\otimes$ to denote join operation):

$$A \otimes B = \bigcup_{i=1}^{n} A_i \otimes B_i$$

Consider first single-processor hash-based join. One of the earliest such algorithms is the GRACE join [18]. It has two phases. The partition phase partitions

two relations into buckets and writes them back to disc. The join phase builds hash tables for buckets from one relation and probes the hash table using tuples of the corresponding buckets from the other relation. The hash function used for partitioning into buckets may also be used for building and probing hash tables during the joining phase. However different hash functions would minimise collisions and is recommended [18].

Many variants of this have been developed. One elaboration is to partition the relations on-the-fly with the join. That is, tuples are read from the first (build) relation and hashed. According to predetermined hash values, qualifying tuples are used to build a hash table i.e. tuples belonging to the current partition are retained and inserted into the hash table. Non-qualifying tuples are discarded [39], written back as sequential files, or written back as bucket files [40]. The same is done with the second (probing) relation. This is done until all hash buckets are processed.

As mentioned, this algorithm is well suited to parallelisation. In shared memory systems, during the partitioning phase, all processors can read pages of the two relations relation and create their hash buckets. During the join phase, all processors can co-operate to build a shared global hash table by reading in parallel pages from a hash bucket of the first relation. After the table is built, all processors read different pages from the corresponding bucket of the second relation, and probe the hash table for matching tuples. All uni-processor hash-based join algorithms may be parallelised in this way. Hash-based join algorithms for shared memory architectures are discussed in detail in [41].

In shared nothing systems the base relations are partitioned across the nodes. Generally, there is no correlation between the node where a tuple is stored and the values of its join attributes. Therefore, in the partition phase, each node reads tuples from local fragments of the first relation and applies a pre-determined hash function to determine a node that the tuple is to be routed to. The second relation is redistributed in

the same way. After the partition phase each node will have all the buckets to be processed by it stored locally. The join phase is a uni-processor join at each site. In [42], hash-based join algorithms for shared nothing systems are studied in detail.

The effectiveness of parallel hash-based join processing is determined by how evenly the load can be divided among the processors while keeping minimal the co-ordination and synchronisation. One factor that makes this difficult is the presence of *data skew*. In real databases it often happens that some values of attributes occur more frequently than others. This is a feature of the data, and is independent of the access patterns. This type of skew may cause substantially different amounts of data to be assigned to different system processors. This can lead to the under- or over- utilisation of processors, which impairs speedup.

2.3.4 Query optimisation and scheduling

Query optimisation refers to the process of selecting an optimal execution plan for a query from a set of feasible plans, based on some objective such as minimising the query response time. One part of this process is to determine a suitable order in which to perform operations. For example, in a query consisting of a number of join operations, it may be beneficial to perform highly selective joins first, thus eliminating a large number of tuples early and reducing the size of intermediate relations. Another part of the optimisation process is concerned with choosing what algorithms to employ for the relational operations within the plan. As mentioned, a join may be performed using a hash-based, nested-loop based, or merge-sort based method; a scan may be a full table one, or involve an index.

A plan can be represented as a tree, with types of nodes determined by the type of algorithms chosen. The shape of the tree and the chosen algorithms determine the amount of parallelism that can be achieved. Consider a query that joins four relations with a hash-based join method and aggregates the result. Let the join use the left

operand (relation or tuple stream) to build the hash table and the right to probe the hash table. A *left-deep* tree for the query is given in Figure 2-4(a). Since a hash table must be built before it can be probed and the result of the join at one level is used to build the hash table of the next level the query is restricted to execute in a sequence of steps, as follows:

1. scan(A)-build(A)
2. scan(B)-probe(A,B)-build(A⊗B)
3. scan(C)-probe(A⊗B,C)-build(A⊗B⊗C)
4. scan(D)-probe(A⊗B⊗C,D)-aggregate(A⊗B⊗C⊗D)

Pipeline parallelism may be exploited within individual steps of execution e.g. the scan, probe and build in the second step may be pipelined. However, no two scans are executed concurrently.

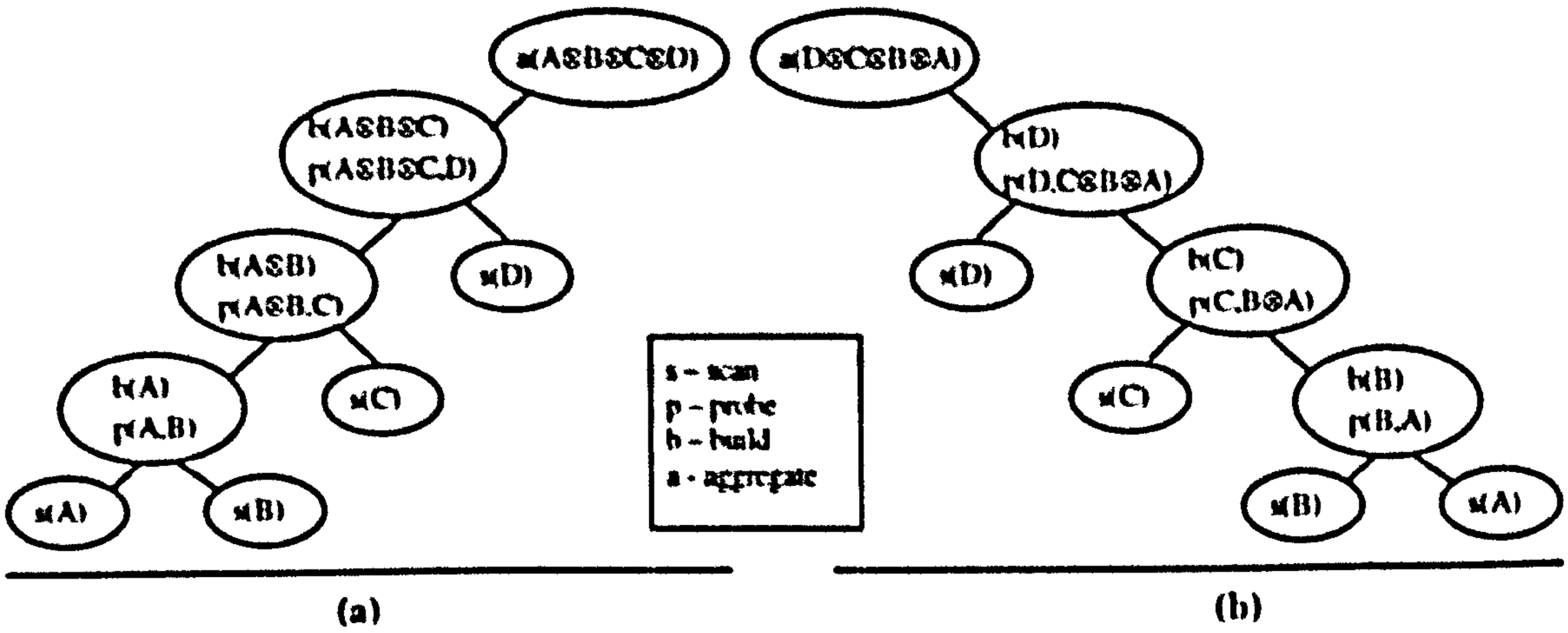


Figure 2-4 Left-deep and right-deep trees

Figure 2-4(b) shows a *right-deep* tree for the same query. With this tree a higher degree of concurrency may be achieved, since the hash table at each level is built from a base relation and not from the result of the previous join. Thus the following execution sequence may be used:

1. scan(B)-build(B), scan(C)-build(C), scan(D)-build(D)
2. scan(A)-probe(B,A)-probe(B⊗A,C)-probe(C⊗B⊗A,D)-
   aggregate(D⊗C⊗B⊗A)

Note that in the first step all scans can be carried out in parallel, provided the three tables are located on separate nodes. In the second step a single pipeline performs the probe phases of all three joins.

Besides determining an optimal tree, the optimisation process must also determine the scheduling and allocation of system resources, such as processors and memory, to concurrent operations. The resource-scheduling phase of query optimisation may be carried out after an optimal tree for the query has been selected. It will ensure an optimal allocation of resources for the chosen tree. Alternatively, resource-scheduling considerations may be taken into account during the generation of the tree. The rationale is that a generated plan may not be optimal without considering resource allocation. Query optimisation and scheduling in parallel database systems is discussed at length in [18, 33, 43, 44].

2.4 Evaluation of parallel relational databases

This section addresses the question of how different designs are to be evaluated relative to each other with the help of benchmarks. Database performance benchmarks [45] can usually be classified into two types. *Transaction processing benchmarks* are designed to test OLTP (on-line transaction processing) operations, typical of applications whose database operations are pre-defined and grouped in transactions. *Decision support benchmarks* are designed to test ad hoc querying on specially constructed databases. The TPC-C transaction processing benchmark and the AS^{3}AP decision support benchmark are described here as examples of each class.

2.4.1 TPC benchmarks

The goal of TPC (Transaction Processing Council) benchmarks is to provide users with a fair comparison between transaction processing systems regardless of hardware and software architectures and operating systems. TPC-C [46, 47] exercises the database components necessary to perform tasks associated with transaction

processing environments emphasising update-intensive database services. Some read-only queries are also included in the benchmark.

The workload is centred on the activities of a wholesale supplier. The company operates out of a number of warehouses supplying relative sales districts. Each warehouse supplies 10 sales districts and each district serves 3000 customers. Each warehouse attempts to maintain stock for 100,000 items in the company's catalogue. In reality 10% of orders must be supplied, in part, by another warehouse (because they are unavailable in the local warehouse). The benchmark is designed to scale as the business grows. The number of warehouses is the basic unit of scaling in TPC-C.

There are 5 transactions available to operators. The benchmark has chosen the transactions and their frequencies to be representative of the activities of the company. Each transaction executes part of the processing of an order, from its entry to its delivery.

The two most frequent transactions are New-Order and Payment. The former places a new order through a single transaction and portrays a middleweight, read-write transaction. The latter updates the customer's balance, with the payment reflected in the district and warehouse's sales statistics, and is a lightweight, read-write transaction. Both transactions have stringent response-time requirements. The other three transactions are Order-Status (queries status of customer's most recent order), Delivery (processes 10 new orders within 1 transaction), and Stock-Level (determines the number of recently sold items with a low stock level).

Ten relations are involved: *Warehouse*, *District*, *Stock*, *Items*, *Parts*, *Customer*, *Order*, *New-Order*, *Orderline*, and *History*. The tables cover a range of cardinalities, from a few rows up to millions of rows. Relationships between table sizes must be maintained according to the scaleability requirements of the benchmark, which is governed by the number of warehouses.

The benchmark simulates company work through the activity of 10 emulated terminals connected to each warehouse. Transactions of all five types are available at each terminal, and their appropriate mix is maintained by selecting a transaction type from a weighted distribution. The benchmark requires that the minimum percentage of each transaction type in the mix be as follows: Payment–43%, Order-Status–4%, Delivery–4%, Stock-Level–4%. There is no requirement for the New-Order transaction as its measured rate is the reported throughput.

On each terminal, a transaction is chosen, executed (keying time is also emulated), and its response time is recorded. This is followed by some think time, and the cycle is repeated. For each type, 90% of the transactions must be completed within specified response time constraints, which are 20 seconds for the Stock-Level transaction and 5 seconds for all other types. The response time constraints are set to give predominance to New-Order as the performance limiting transaction. The response time of a transaction is measured as the time elapsed between the launch of the transaction and the receipt of the last of any data returned by the transaction.

The reported measure of performance is the number of orders processed per minute, computed as the total number of completed New-Order transactions during the measurement interval, divided by the elapsed time of the interval. This is all subject to the response time constraint. The measurements of throughput should be taken when the system is in a steady-state condition, which would represent the true sustainable throughout of the system.

2.4.2 AS^{3}AP benchmark

Another benchmark for relational database systems is the AS^{3}AP (ANSI SQL Standard Scaleable and Portable) benchmark [48]. This benchmark is not defined in the context of business environment, as TPC-C is, and is intended to take into account more of the functionality that modern DBMSs provide.

The benchmark consists of two types of tests. The *single-user tests* consist of utilities for loading and structuring the database, and queries designed to test access methods and basic query optimisation – selections, simple joins, projections, aggregates, one-tuple updates, and bulk updates. The *multi-user tests* model different types of database workloads: OLTP workloads, information retrieval (IR) workloads, and mixed workloads. The single-user test set is designed to exercise the basic functionality that relational DBMSs provide, while the multi-user tests measure the DBMS performance as a function of the workload profile.

The measure of performance defined by the benchmark is the *equivalent database size*, defined as the maximum size of the AS^{3}AP database for which the system is able to perform the complete set of single- and multi-user tests in under 12 hours. On the basis of this measure, two additional measures are defined. The *cost per megabyte* is the total price of the DBMS divided by the equivalent database size. The *equivalent database ratio* for two systems is the ratio of their equivalent database sizes.

The only measurement required by the AS^{3}AP benchmark is for the elapsed time to be within the 12-hour limit. Even though additional measures e.g. on CPU or disc utilisation, may be taken, they are not required by the benchmark. In order to get high rating on this benchmark vendors must estimate the maximum size of the test database that will enable them to meet the 12-hour requirement. The queries are designed to scale with the database size.

All benchmark queries from a run are embedded in a host program. For multi-user tests, a number of processes are forked concurrently, each running a simple script. The number of processes is equal to the number of users, which is determined by dividing the logical database size (in megabytes) by 4. No terminal emulator is needed, and no streams of job arrivals are generated. There is no simulation of think time.

The benchmark defines 5 tables. The first one, called *tiny*, has one attribute and one tuple used only to measure overhead. The four main relations are: *Uniques*, where

all attributes have unique values; *Hundred*, where most attributes have exactly 100 unique values; *Tenpct*, where the unique values of an attribute represent 10% of the total number of values; *Updates*, which is customised for updates, with different distributions used and three types of indexes.

The main relations have the same 10 attributes with the same names and types. Moreover, they all have the same average tuple width (100 bytes on average) and the same number of tuples. An additional attribute is provided (fill) to ensure the 100-byte requirement holds. The 10 attributes are given in Table 2-1. The experiments used in Chapter 7 to validate the response time model consist of queries over the *Uniques* table.

Attribute name	Attribute type	Length
key	Integer	4
int	Unsigned integer	4
signed	Signed integer	4
float	Floating point	4
double	Double precision	8
decim	Exact decimal	18, 2
date	Datetime	8
code	Alphanumeric	10
name	Character string	20 fixed
address	Variable length string	2 to 80 (20 avg)
fill	Character string	as needed

Table 2-1 Attributes of the AS^{3}AP relations

2.5 Commercial products and systems

This section discusses three commercial products: the Goldrush MegaServer parallel platform from ICL and two relational database systems implemented for it, the Informix Extended Parallel Server and the Oracle Parallel Server with Parallel Query Option. In the beginning of 1996 Goldrush/Oracle and Goldrush/Informix were the state-of-the-art shared disc and shared nothing high performance database solutions offered by ICL. Under the MERCURY project [49] the three systems were studied in detail and provided many of the ideas that developed into the performance models described in the thesis.

2.5.1 ICL Goldrush MegaServer

The ICL Goldrush MegaServer is an open, parallel database server designed for high performance and availability. The architecture of the system is presented in detail in [50, 51, 52] and a summary of these papers is given here.

The hardware architecture of the Goldrush system is shown in Figure 2-5. It consists of a set of up to 64 Processing Elements (PEs) and Communication Elements (CEs), and a single Management Element (ME), connected together by a high performance network (Deltanet).

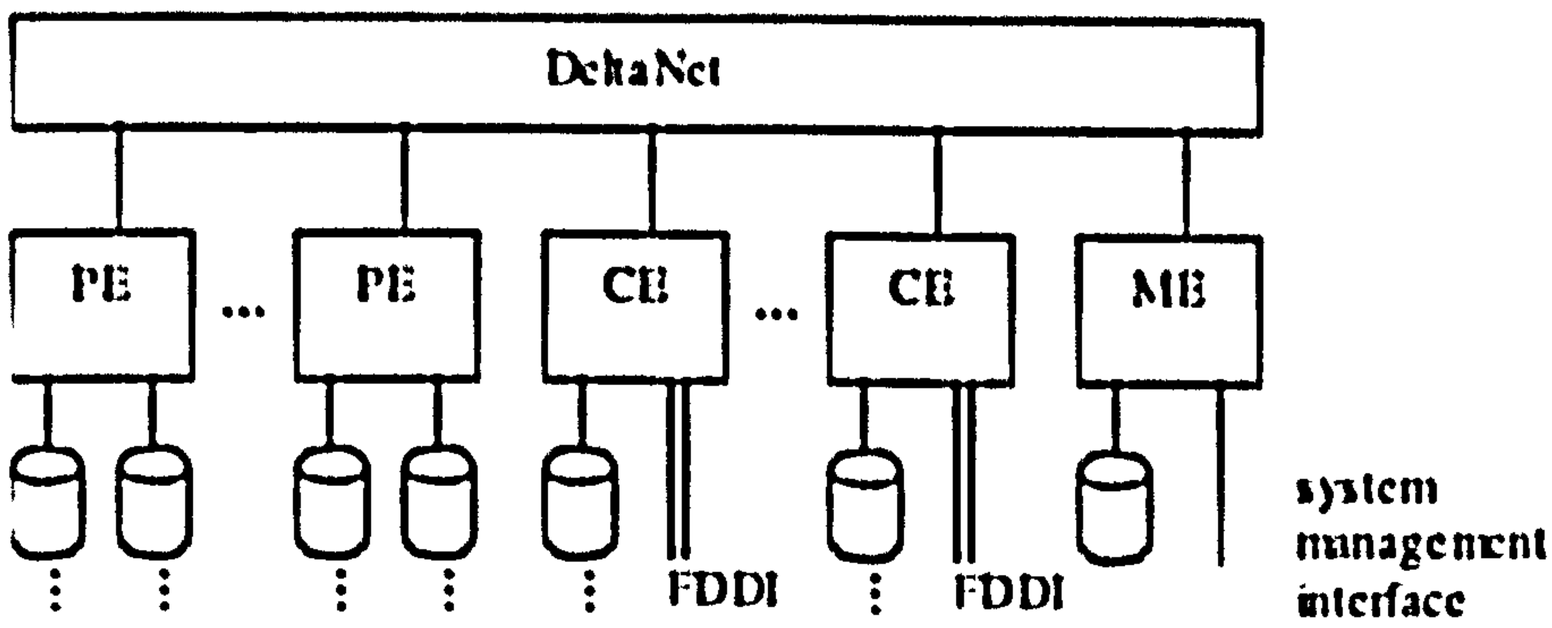


Figure 2-5 The Goldrush architecture

Each PE runs a Chorus micro-kernel based SVR4 UNIX operating system and database back-end servers. It consists of two 90MHz HyperSPARC microprocessors: a PU (Processing Unit) running the database server and a SSU (System Support Unit), responsible for message passing over the Deltanet. Up to 256 Mbytes of RAM store is provided in the PE, primarily for use as database cache. Each PE also has two fast and wide SCSI-2 connections, allowing up to 30 discs to be connected.

The CE is identical to a PE except that two FDDI couplers for client connections replace one of the SCSIs. The CEs do not run database servers, but are dedicated to relaying messages transparently between the PEs and FDDI couplers. It is typical to have 1-2 CEs per 16 elements. The ME is a conventional mid-range UNIX processor (an ICL DRS6000) which runs Goldrush management software.

The Deltanet is a multi-staged network built from the basic building block of an 8×8-crossbar switch, which provides unidirectional channels from each of its 8 inputs to

any of its 8 output ports. By connecting a number of switches in stages, any number of elements (up to the maximum 64) can be interconnected. Packets are 128 bytes long. If two or more packets arrive at the inputs of a switch simultaneously, and each needs to be routed through the same output, contention results. The switch has 4 buffers per input, so one of the packets is retained until the output is free. The Deltanet runs asynchronously with the elements and allows transfer rate of up to 25 Mbytes per second each way per element.

Goldrush relies entirely on message passing for inter-PE communication. This is done efficiently through a lightweight communications protocol. It is available through UNIX interfaces so that software can exploit it. For example, the Informix and Oracle software use it for communicating when exploiting intra-query parallelism.

For database systems supporting a shared disc architecture, Goldrush provides a Global Coherent Filestore (GCF). Even though physically each disc is attached to one element only, each element creates its own portion of the GCF on its own local discs and then cross-mounts the global file store from all other elements, thus making it accessible to the database server running on it. The GCF is implemented in the OS kernel and uses the lightweight communications protocol.

A Distributed Lock Manager (DLM) is implemented to provide coherency/concurrency control for database servers running within a shared disc system. It is distributed across all PEs, with an instance running on each one. Each global lock is managed by one instance of the DLM and all requests for it are sent to that instance by the PE generating the request. All communication with the DLM is by the lightweight communications protocol.

Goldrush can be configured as either a shared disc or a shared nothing system. In a shared disc configuration each PE runs its own instance of the database. The tables are placed across discs attached to a number of PEs, with the GCF allowing shared access. Each PE maintains a local database cache in memory and the DLM is used to

ensure coherency. In a shared nothing configuration, again, each PE runs a database server and the tables are partitioned across the PEs. Each PE only accesses the table fragments stored locally. Queries are decomposed into 'fragments' and directed to PEs which own relevant data. There is no need for the services of the DLM, as no global coherency needs to be maintained.

2.5.2 Informix XPS

In 1994, Informix released Informix OnLine version 7.1, which used the parallelism built into its Dynamic Scaleable Architecture (DSA), to take advantage of SMP platforms. In 1995, DSA was extended to take advantage of MPP platforms and Informix OnLine Extended Parallel Server (XPS) version 8 was released [27].

On Goldrush, Informix XPS implements a shared nothing database architecture. Each PE runs its own instance of the database server (a *co-server*) which consists of basic Informix XPS services for managing its own logging, recovery, locking, and buffer management. Each co-server owns a set of discs and the partitions of the database that reside on them. The DLM of Goldrush is not used.

Tables can be partitioned into fragments, which can be allocated to different nodes. Fragmentation can be hash based, round robin based and expression based. Expression based fragmentation allows a user to define an arbitrary rule. The type of fragmentation and the physical location of the fragments are specified at table creation time. For sequential table or index scans, Informix XPS can be configured to read several pages ahead while the current pages are being processed.

A co-server interacts and co-ordinates activities with other co-servers in order to execute queries in a highly parallel fashion, utilising both inter- and intra- operator parallelism. A request manager resides on each co-server and manages the execution of incoming queries through interaction with other XPS services. These include: a cost-based query optimiser; a metadata manager, which determines how the data is

distributed and where it resides; and a scheduler, which parallelises the execution plan and distributes query sub-tasks among co-servers.

Operators within execution schedules are processed in parallel by threads within a single co-server and across co-servers. Operators that may be scheduled to execute in parallel include scanning both data and indices, join, sort, insert, update and delete, and set and aggregation operations. A special type of operator, named *exchange*, is used to enable parallel processing. It corresponds to the split and merge primitives described in Section 2.3.1. Through an exchange operator the output tuple streams of one or more operators may be repartitioned before being directed to the next set of operators in the schedule. The exchange operators support pipelining of intermediate results. Informix XPS inserts exchange operators at places within an execution schedule where parallelism is beneficial. In addition, support operations such as loading and unloading of data, backups, archiving and restoring are also designed to run in parallel.

The most frequently used join method is the hash join. If the memory fills up during the building phase, the biggest hash bucket is moved to disc. If a probing tuple from the second relation matches the bucket moved to disc, it will also be written to disc. When all tuples matching memory-resident buckets have been processed, the disc-resident buckets along with the probing tuples are read into main memory again to complete the join operation. An example Informix XPS execution plan is shown in Appendix A.

The buffer management used by Informix XPS is described in Chapter 8, where the Informix cache model is presented. Other important aspects of the system, such as low-level thread co-ordination, logging, and buffer flushing are discussed at length in [53].

2.5.3 Oracle Parallel Server

The Oracle Parallel Server [7] consists of separate Oracle 7.2 *instances* running simultaneously on Goldrush nodes. A single Oracle instance runs on each node. A shared disc architecture is established through the use of a technology called parallel cache management. Thus, each instance maintains a separate database cache and set of background processes, while all instances share the same data files and control file. All instances can execute transactions against the same database concurrently and each instance can have multiple users executing transactions.

A single Oracle instance consists of several physical structures, memory structures and processes. These include the System Global Area (SGA) cache for database pages and log entries; the SGA shared pool for SQL statements; the background and server processes; and different types of files - database files, control files and log files.

Requests from connected user processes are handled by *server* processes, which parse and execute SQL statements, read database pages into cache of the SGA, and return the results to user processes. A single server process may work for many users. The most important *background* processes include: DBWR, which manages the database buffer cache by writing all changed buffers to the data files; LGWR, which writes the information in the log buffers to disc; PMON (process monitor) and SMON (system monitor), which reclaim database resources that are no longer needed. Server and background processes run as OS processes.

An Oracle database is subdivided into smaller logical areas of space known as *tablespaces*, each of which contains one or more data files. Tablespaces may be composed of data files created on separate discs and partitions. This allows for a basic form of data fragmentation. However, there are no facilities for more sophisticated fragmentation and placement, similar to those in Informix XPS.

An Oracle instance may be configured with an option called *Parallel Query*, which supports intra-query parallelism including both inter- and intra-operator parallelism. This is achieved by *slave* processes, which are co-ordinated by a *master* process that divides up the work and combines the results. When properly configured, the slaves will access different sections of tables, on different physical discs. For a query to be parallelised it must begin with a full table scan.

Slaves reading from a table on disc or in memory are called producers. Producers can apply simple functions to rows (e.g. predicate checks), but do not process multiple rows together. Consumers are slaves used to apply more complex functions. For example, in a sort, the producers read the data from the table and filter on any given condition. Then they send the matching rows to different consumers based on the value being sorted on, thereby partitioning the rows across the consumers. The consumers sort the rows passed to them, and pass the sorted rows back to the master, which merges their output together.

The parallel cache management mechanism used within the Oracle Parallel Server, which uses the DLM of Goldrush, is described in Chapter 8, where the Oracle cache model is presented. In [53] other aspects of the system, including locking, logging, and buffer flushing are discussed in more detail.

2.6 Summary

This chapter has reviewed the field of parallel relational database systems and introduced some concepts and terminology which are used throughout the rest of the thesis. The three common hardware architectures underlying modern parallel database systems were described and compared. The various ways of utilising parallelism in query execution were discussed and the concepts of operator tree, dataflow graph, pipeline parallelism and partitioned parallelism were introduced. The notions of data partitioning and placement were reviewed briefly. The hash-based parallel join

operation was described. The processes of query optimisation and scheduling, which aim to produce an optimal execution schedule for a query, were briefly discussed.

Two standard benchmarks for database systems were described and the notions of throughput and response time were introduced. The three commercial systems, which have inspired the performance models developed in the thesis, were described.

The next chapter focuses on performance prediction in parallel database systems.

Chapter 3

PERFORMANCE PREDICTION IN PARALLEL DATABASE SYSTEMS

3.1 Introduction

In the database field performance prediction plays an important role. From the point of view of the database system designer it can be used to evaluate different software and hardware design options to optimise performance. Once the system has been developed, it is useful in determining the configuration of a system to meet user requirements as well as in subsequent tuning to obtain improved performance. Tools can also be used to evaluate how performance will vary in the future as changes occur in the volume of data or balance of queries. With parallel database systems, these tasks are even more complex and there is an increased need for performance prediction tools to assist the user.

Performance prediction tools are built around a performance estimation 'core'. This module should be based on a conceptual model (understanding) of how the system actually works. In the context of parallel database systems, the model should take into account aspects such as query execution, scheduling, concurrency-control, buffer management, and so on.

From a model some estimates of performance factors can be derived. In the light of the previous discussion on benchmarks and from the point of view of the system architect evaluating alternatives, performance measures of interest include:

- Transaction response time
- Transaction throughput
- Resource utilisation, queue length and response time

- Bottleneck resource(s)

A performance modelling methodology is used to represent the conceptual model and derive such performance measures from it. Several such methodologies have been developed for the class of hardware/software systems discussed here, namely systems in which software processes or threads share resources and co-operate to accomplish overall system goals, incurring delays due to their contention for resources.

Section 3.2 discusses some methodologies for performance modelling, applicable to parallel database systems as outlined in the previous chapter. Discrete event simulation, queueing networks and performance Petri nets are different methodologies that allow modelling of resource sharing by concurrent processes. Performance evaluation process algebra is a formal approach to modelling system behaviour. The Method of Layers [54] is specifically designed to model modern software systems; process behaviour such as synchronisation with the use of fork/join primitives or the request of service by one process from another, may be represented.

In the context of the previous chapter, Section 3.3 discusses previous work on performance estimation and modelling in parallel database systems. Section 3.4 focuses on the STEADY performance prediction tool. STEADY is the starting point of the work carried out in the thesis. The response time estimation mechanism (Chapters 4, 5 and 6) and cache models (Chapter 8) are developed as extensions to the original STEADY tool.

3.2 Approaches to performance modelling

In this section Discrete Event Simulation, Performance Petri Nets, Performance Evaluation Process Algebra (PEPA), Queueing Networks and the Method of Layers are reviewed.

3.2.1 Discrete event simulation

The use of discrete event simulation [55] is a modelling methodology, which allows system behaviour to be described very accurately and is applicable to a wide range of problems. Within a simulation program, system behaviour is usually modelled by a main control loop. Each execution of the loop represents a single event from a range of possible events taking place. The simulation time clock is advanced by the amount of time since the last event. The objective of the simulation run is to obtain a set of experimental observations that are characteristic of the actual, underlying system. There are two basic approaches to acquiring statistics during a simulation run: a histogram may be built up by counting observed values, or running sums may be maintained to calculate statistics directly.

Even though discrete event simulation allows system behaviour may be described in considerable detail, the computation time and resources needs are usually rather high. This is incompatible with the requirement for a practical tool capable of rapid estimation of performance. Therefore, less computationally intensive analytical methodologies were considered more appropriate for the performance prediction task, and are discussed in more detail in following sub-sections. Discrete event simulation was used, however, to assess the heuristic rule developed in Chapter 5.

3.2.2 Performance Petri nets

Petri Nets provide a general notation for representing concurrent process execution. A Petri Net is a collection of *places* (denoted by circles) and *transitions* (denoted by bars) linked together by directed arcs. *Tokens* (denoted by discs) reside in places and move from place to place via the arcs. This movement is governed by the transitions. In general, transitions represent resources and synchronisation points, while places represent queues and tokens are processes that synchronise and share resources. When a token is available on each input place of a transition, the transition becomes

enabled, fires, the tokens are consumed, and a token appears in each of the output places of the transition. A *marking* of a Petri Net describes the number of tokens in each place. The set of reachable markings, given an initial marking, is the Petri Net *state space*.

An example Petri Net and its state space is given in Figure 3-1 [54]. A simple fork-join system is shown. The single token is initially in place P1 and the parent process is receiving service. An amount of service time is required before the parent can fork two child processes. After the fork, two tokens exist in places P2 and P3, respectively, each representing a child process. Each child receives some amount of service time and waits for synchronisation. Once both have completed service, synchronisation occurs and control is passed back to the parent.

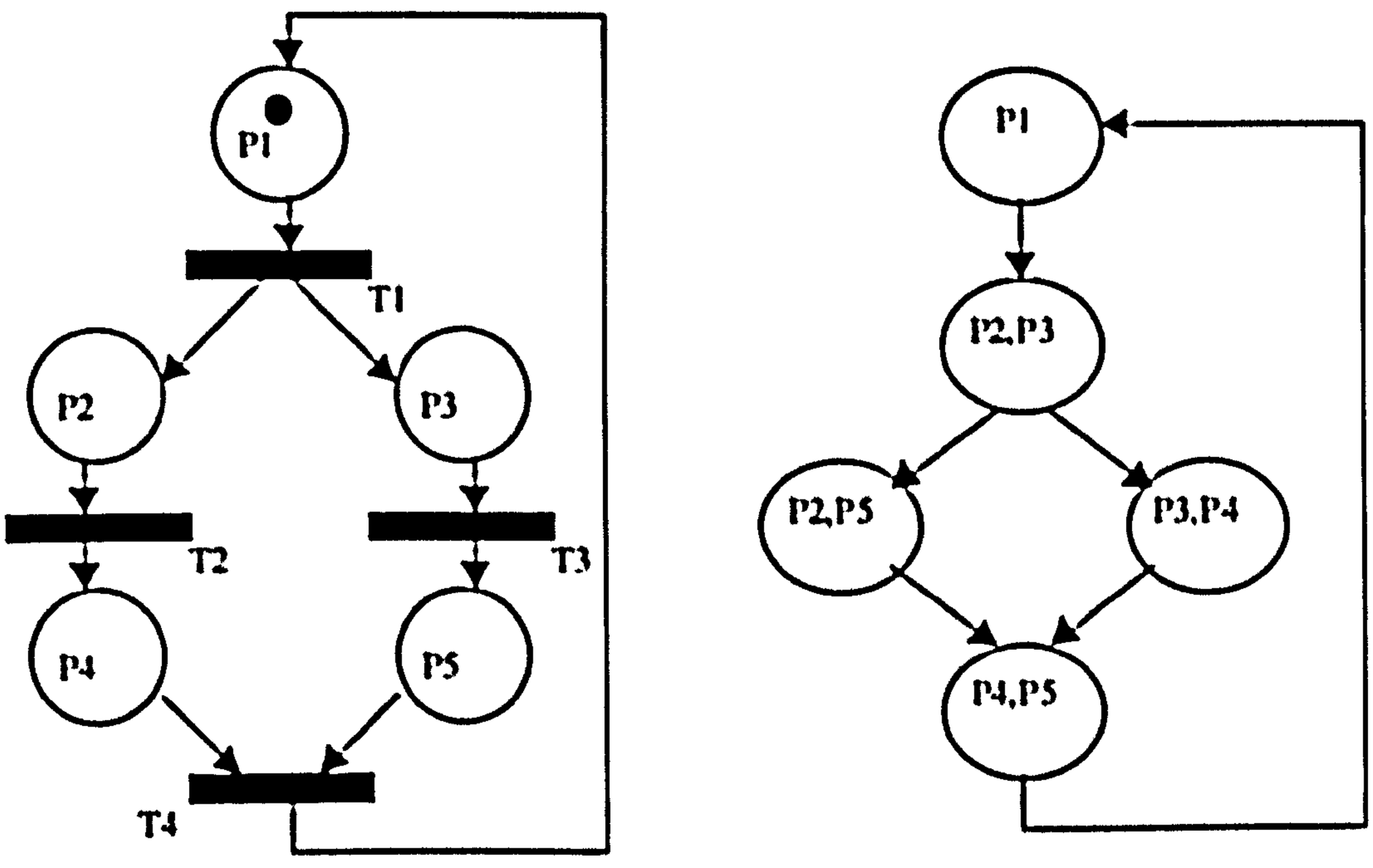


Figure 3-1 A Petri Net and its state space

Petri Nets in which transitions are associated with time (as the one shown in Figure 3-1) are termed Performance Petri Nets. After a transition is enabled, some time (a sample from a given service time distribution for the transition) passes before the transition fires. Stochastic Petri Nets (SPN) [56] and Generalised Stochastic Petri Nets (GSPN) [57] are two examples, both capable of representing synchronisation and

timing. SPNs have exponentially distributed transition firing times, while GSPNs also allow immediate transitions.

A system of linear equations can be constructed that uses timing information to determine the rate of movement from one state to the next. The equations are solved to obtain the steady state probability of residing in each state [56]. From these probabilities, performance measures for the net may be found. These include resource utilisation, queue lengths, and customer response times. In general, the solution requires that the complete state space be enumerated. This means that the solution cost can be prohibitively large for more complex models.

The Petri Net methodology has been used in the past to model aspects of parallel database execution. For example, in [58] timed Petri Nets are used to model the data- and control flow during parallel query execution.

Although there are certain classes of Petri Nets for which efficient solutions exist, or for which efficient approximation techniques exist (some examples of such methods are given in [54]), the Petri Nets approach was not considered suitable for the task at hand. It was anticipated that the state space constructed for the models of parallel query execution would be too large and, correspondingly, the time to obtain performance measures would be unacceptably high. On the other hand, the use of third-party tools to solve large Petri Net models would violate the requirement for a self-contained performance prediction tool.

3.2.3 Performance evaluation process algebra

Process algebras are mathematical theories designed to model communication and concurrent systems. Performance Evaluation Process Algebra (PEPA) [59] is a method developed to investigate the impact of the compositional features of process algebras upon performance modelling.

In PEPA, a system is expressed as an interaction of *components* that engage in *activities*. The components correspond to parts of the system or events in the behaviour of the system. Each component has a behaviour, which is defined by the activities in which it engages. Every activity has an action type and an associated duration (represented by an *activity rate*), and is written as (α, r) where α is an action type and r the activity rate.

PEPA has a small but powerful set of combinators used to model system behaviour. The sequential composition, $(\alpha, r).P$, is the basic mechanism by which the behaviour of a component is constructed. This specification means that the component will perform activity (α, r) and behave as P on completion.

The selection or choice composition, P + Q, represents a system which may behave either as P or as Q but not both P and Q at the same time. Both components are enabled. The co-operation or parallel composition, P <L> Q, denotes the fact that P and Q can proceed independently and concurrently with any activity whose action type is not contained in L. However, for any activity whose action type is included in L, P and Q must synchronise to achieve the activity. In this case one component may be blocked waiting for the other one. Finally, the constant A is a component whose meaning is given by the defining equation (A = P) which gives component A the behaviour of component P.

To evaluate a PEPA model, a state space is constructed, following the semantics of the PEPA combinators. The corresponding state transition rate matrix of the state space is formed. A system of linear equations is then solved to obtain the steady state probability of each state, from which performance measures for the system can be computed. PEPA models suffer from state-space explosion [59] when the models become complicated. To overcome this problem, different notions of equivalence may be applied to simplify a model by replacing a set of co-operating components with an

equivalent lumped component. It can be proved that the new model has equivalent behaviour, while having a reduced state space and, hence, a more efficient solution.

In [60], Pua studies the applicability of PEPA as a modelling methodology for parallel database systems. PEPA models of simple transactions are built to account for inter- and intra- query parallelism during transaction execution.

The PEPA methodology allows the formal modelling of complex system behaviour. However, it was not chosen in this study for the performance modelling of query execution in parallel database system. It was expected that the constructed models and corresponding transition rate matrices would be unacceptably large. Previous experience [60] indicates that even a relatively simple model may have a state space consisting of several thousand states, while complex models may lead to millions of states. To solve the transition rate matrices of such models one must use mathematical packages (e.g. Matlab [61]), but at present there are limits to the size of matrices they can handle. Moreover, relying on such a package for the underlying computation is not compatible with the requirement for a stand-alone performance estimation tool.

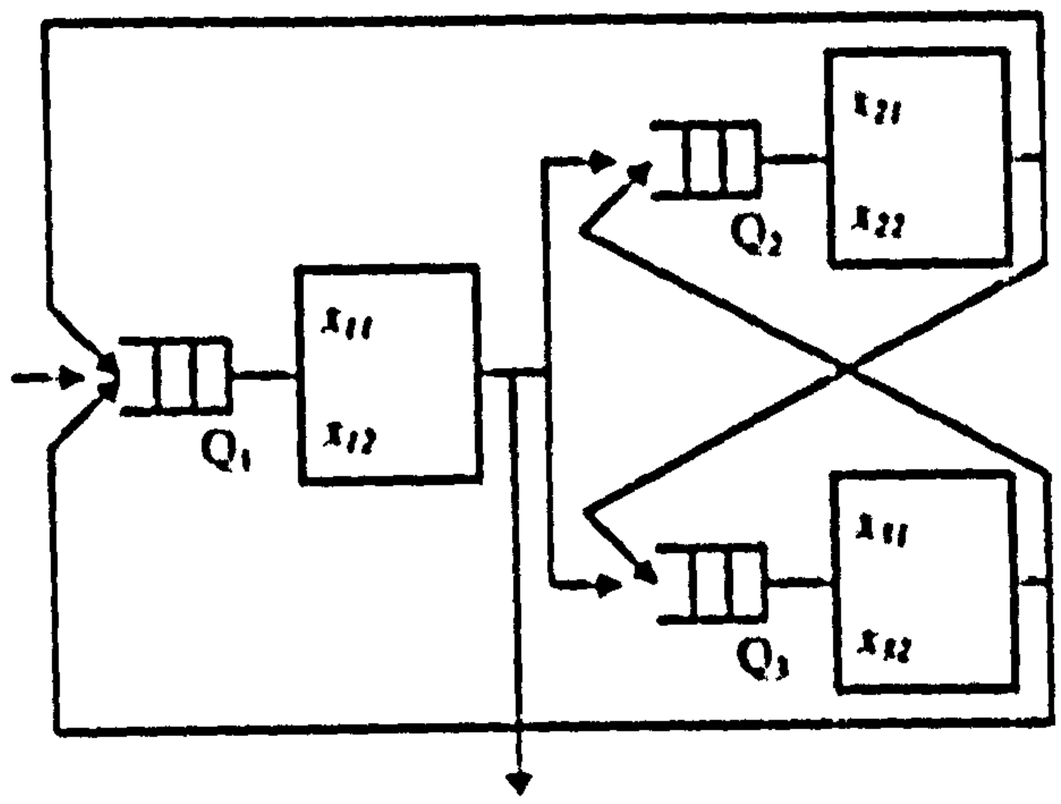
The use of transformations, based on notions of equivalence, may be used to reduce a model to a simpler one, equivalent to the original. It is not clear, however, whether this process can be automated. Moreover, even the equivalent simple model may prove too big to be practical to work with.

3.2.4 Queueing network models

Computer systems are often modelled as networks of queues [55, 62, 63, 64]. For example, a program may be queued at a disc while attempting to complete an I/O operation. Queueing networks can be used to describe the competition of concurrent processes for shared resources.

A queueing network model consists of *servers*, *customer classes*, and a *description* of how the classes use the servers. A customer class contains one or more customers (corresponding to processes) that have independent and statistically identical behaviour. A class is *closed* if there is a fixed number of customers of the class in the network at all times. No customers of a closed class enter or leave the network. If customers are better described as arriving at some rate, satisfying their service requirements, and leaving the system, the class is an *open* class. A network consisting only of closed classes is a *closed queueing network*. If only open classes are defined, the network is an *open queueing network*. If both open and closed classes are present, the network is a *mixed queueing network*.

In the network, customers are processed at a queue according to the queueing discipline and, after receiving service, proceed for service to the next queue. The description of customer movement through the network is given by the *routing probabilities* of each customer class. The routing probabilities are of the form $p_{ir,js}$ which is the probability that upon completion of service at queue i a customer of class r moves to queue j for service as customer of class s. The probabilities of departure to and arrival from the outside are denoted as $p_{ir,0}$ and $p_{0,js}$. An example of an open network with two classes and three nodes is given in Figure 3-2 [63]. The routing probabilities of both classes are shown. There is no class switching for this example network. Service times are assumed to be exponentially distributed and their means, x_{ir} are also shown. Inter-arrival times are also exponentially distributed with rates λ_1 and λ_2, equal to 1 request per second.



$p_{01} = 1.0$ $p_{21,0} = 0.6$
$p_{12,1} = 0.4$ $p_{21,1} = 0.4$
$p_{12,2} = 0.1$ $p_{21,2} = 0.1$
$p_{12,3} = 0.1$ $p_{21,3} = 0.1$
$p_{13,0} = 1.0$ $p_{22,1} = 0.7$
$p_{23,2} = 0.1$ $p_{22,2} = 0.1$
$p_{23,3} = 0.6$ $p_{22,3} = 0.4$
$p_{32,0} = 0.1$ $p_{32,1} = 0.6$

$x_{11} = 0.125sec$ $x_{12} = 0.042sec$
$x_{21} = 0.083sec$ $x_{22} = 0.031sec$
$x_{31} = 0.083sec$ $x_{32} = 0.028sec$

$\lambda_1 = 1\ sec^{-1}$ $\lambda_2 = 1\ sec^{-1}$

Figure 3-2 Simple open queueing network

3.2.4.1 Product-form networks

Given a queueing network, performance measures such as the mean number of customers of a class at a given queue or the mean response time of customers from a class, may be computed. One can, in principle, always attempt to solve so-called global balance equations for a queueing network in order to obtain the steady-state probabilities of all possible states of the network. The mean values of all important performance measures of the network can be calculated from these. However, for large networks this technique is very expensive because the number of equations can be very large [63].

Therefore, networks whose solutions can be obtained efficiently without generating the underlying state space are of particular interest. There is a special class of networks with the property that there exists a unique solution to the so-called *local-balance equations* of the system. Such networks are said to have the *local balance property* [63]. The local balance property implies that the state probability of the whole system is equal to the product of the state probabilities of the individual queueing centres of the network. This means that each queueing centre can be examined in isolation from the rest of the network in order to obtain its performance measures. Such networks are known as *product-form* or *separable* networks.

Whether or not a queueing network has local balance depends on the properties of the queueing centres that the network is composed of. The celebrated BCMP theorem [65], extending earlier results by Jackson [66] and Gordon and Newell [67], specifies the properties that individual queueing centres must possess in order for the queuing network to have local balance, and hence, a product-form solution. For example, the theorem permits FCFS (first come, first served) queueing centres to be used. However, they are restricted to have exponentially distributed and class independent service times. If more than one class is considered, a different queueing discipline, such as PS (processor sharing) must be assumed. The queueing centres from the example network are assumed to be PS queues.

Solving open product form networks with the BCMP theorem is straightforward and consists of three steps.

1. First, the *visit ratios* of individual queues are obtained by solving the traffic rate equations of the network. The visit ratio e_{ir} is defined as the mean number of visits of a customer of class r to queue i for each arrival of class r customer from outside. The traffic rate equations can be written as follows:

$$e_{ir} = p_{0,ir} + \sum_{j=1..N} \sum_{s=1..R} e_{js} p_{js,ir}$$

For example, for class 1 from the network given in Figure 3-2, the visit ratios are computed from this system of equations as follows:

$$e_{11} = p_{0,11} + e_{11}p_{11,11} + e_{21}p_{21,11} + e_{31}p_{31,11}$$
$$e_{21} = p_{0,21} + e_{11}p_{11,21} + e_{21}p_{21,21} + e_{31}p_{31,21}$$
$$e_{31} = p_{0,31} + e_{11}p_{11,31} + e_{21}p_{21,31} + e_{31}p_{31,31}$$

Solving this gives $e_{11} = 3.33$, $e_{21} = 2.29$, $e_{31} = 1.92$. Similarly, the visit ratios for class 2 are $e_{12} = 10$, $e_{22} = 8.05$ $e_{32} = 8.42$.

2. Second, the utilisation of each node is computed using the service time and visit ratio, according to:

$$\rho_i = \sum_{r=1..R} \rho_{ir} \text{, where } \rho_{ir} = \lambda_r e_{ir} x_{ir} \qquad (3.1)$$

Continuing with the example, $\rho_1 = 0.83$, $\rho_2 = 0.44$, $\rho_3 = 0.35$.

3. Finally, the performance measures of interest, such as the mean number of customers from a class at a given node are calculated. For example, the mean number of customers at each node may be computed according to:

$$K_{ir} = \frac{\rho_{ir}}{1 - \rho_i}$$

Taking as example class 1 and node 1, K_{11}=2.5.

If class switching is allowed, the application of the BCMP theorem is slightly more complicated [63].

3.2.4.2 Non-product-form networks

Although the BCMP theorem allows the modelling of complex systems for which performance results may be obtained efficiently, most practical queueing problems lead to non-product-form networks. Some examples are:

- networks of queues with non-exponentially distributed service times;
- networks where some job classes have priority over others;
- networks where a customer may occupy two or more resources simultaneously (e.g. a computer program uses the memory and processor at the same time);
- networks that model parallel processing and synchronisation in computer systems, e.g. the use of fork-join constructs.

Various methods have been developed to approximate the exact solutions of networks with different types of non-product form behaviour. One approach is to decompose a model into a number of sub-models, which are assumed to be in product form. The results from sub-models are combined to provide performance estimates for the original model. For example, [68] considers parallel processing in fork-join systems, which are modelled with open queueing networks. In the approximate analysis, the sub-network that contains the fork-join construct is replaced by a

composite load-dependent node. The service rate of the composite node can be determined by analysing the sub-network in isolation. In [63] an extensive review of various approximation methods is provided.

The advantage of queueing networks over other modelling methodologies is that the solutions of queueing network models can be found efficiently. Unfortunately, if product-form networks are used, there are restrictions on the type of behaviour that can be represented. Modelling the different types of dependencies and synchronisation that take place during parallel query execution represents non-product-form behaviour and demands the use of complex approximation techniques. Nevertheless, a queueing networks approach (discussed in detail in Chapters 5 and 6) has been chosen in preference to the other methodologies. The approach deals with two forms of non-product-form behaviour that arise (non-exponential service times and synchronisation), yet uses relatively simple analytical formulae. Thus, the approach is not as computationally intensive as other methods and does not lead to unacceptable delays in producing performance estimates.

3.2.5 Method of layers

The Method of Layers [54, 69, 70] is designed to capture the behaviour of systems of co-operating processes within modern multi-processor computers. For example, in parallel database systems, the degree of parallelism within a process may change. Also, parallel threads synchronise and communicate while sharing resources. It is also possible for threads to act as both customers and servers in relation to other threads.

In the method of layers the requests for service amongst processes are described as a graph, with processes that do not request service from other processes at the lowest levels and other processes at higher levels. The mean response times of processes at one level of the model are estimated by viewing the processes at the next lower level as

the servers of a closed product-form queuing network. Processes that request service are considered as customers and those that provide service are servers. This representation captures the possible queueing delays incurred by the processes requesting service if the serving process is busy doing work on behalf of other calling processes. A second queuing network model is used to determine queueing delays at physical devices. Each process in the system is represented as a customer, and each device as a server. The results of the software and device contention models are combined to provide performance estimates for the system. The two models are solved alternately, with the solution of one helping to determine input parameters of the other.

The method of layers proposes a useful framework for representing interactions among concurrent threads and published results indicate that it achieves very good accuracy. However, it is not clear what level of detail may be represented. For example, it appears impractical to represent the detailed sequence of resource consumption described by a resource usage profile (see next chapter). In addition, the system behaviour must be adapted to the layered software service paradigm.

3.3 Performance modelling studies in parallel databases

This section gives examples of performance modelling in parallel database systems. First, some approaches to estimating the performance of specific techniques or algorithms, e.g. hash-based join methods, scheduling algorithms, and load balancing mechanisms, are discussed. Next, some more complete models of query execution in parallel database systems are mentioned. Finally, some existing commercial tools with performance prediction capabilities are presented.

3.3.1 Modelling architectures and techniques

A number of researchers who study different database architectures and aspects of parallel database operation, such as skew, scheduling, hash-join algorithms, cache coherency, etc., propose and use models of query execution which produce performance

estimates including system throughput and response time. Different modelling methodologies are used. Some examples are presented below.

An early study on the benefits of the shared nothing, shared memory and shared disc systems is by Bhide [71], where the three architectures are represented as closed queuing systems. Simulated transactions read and modify database pages, cycling between the simulated disc, CPU and network resources, executing according to a two-phase locking protocol. For the shared disc architecture, a cache coherency scheme is also implemented. Transaction throughput and response time are the chosen measures of performance. A number of issues relevant to the shared disc model are investigated, such as the pay-off from maximising lock request locality and the effects of batching and message cost. The study concludes that the shared everything architecture outperforms the other two, but notes that its limitation is the shared bus, which is not modelled in the simulation. A more recent simulation study of the three architectures is [72].

The effects of data skew on performance have been studied extensively [73, 74, 75, 76, 77]. For example, in [77] the effect of data skew on response time of queries using joins, is studied. A taxonomy of types of skew and a modelling methodology are proposed. A method for calculating response time is proposed and is applied to two existing hash join algorithms in the presence of skew. The response time analysis decomposes each algorithm into phases, each of which has a number of steps and can be partitioned across a number of processors. The time of each step is calculated based on the number of tuples involved, and is added to the total time of the resource responsible for the step (disc, CPU or communications). A bottleneck resource is established and its response time is used as the response time of the algorithm phase. The sum of the response times of all phases gives the response time of the algorithm.

In [78] scheduling in shared-nothing systems is studied and an analytical model of parallel execution and resource sharing is developed to estimate the response time of

a given schedule (and, hence, the corresponding query). The model takes into account the overlapped use of multiple resources and captures both partitioned and pipeline parallelism. The response time of a parallel schedule is determined by either the slowest executing operator, or the load at the most heavily congested resource in the system, whichever is greater. Other performance studies of scheduling include [79] (based on simulation) and [80] (based on analytical models).

In [81] a method for efficient execution of multiple pipelined hash joins is proposed. The original execution tree is transformed into an allocation tree, each node of which is a multi-stage pipeline of hash joins. Several different hash-join strategies are studied, all based on the idea of the execution tree. A combination of simulation and analytical techniques is used to derive the execution times of the different schemes. A relatively simple analytical formula is used to compute the execution time within each node. The formula adds up the execution time for each phase of the joining of the relations, namely reading, hash-table building, and probing. The simulation is used to traverse the allocation tree and carry out the join operations in parallel. Many other analytical or simulation-based performance studies of hash-based join algorithms exist [82, 42].

Work on cache coherency policies uses models of response time to quantify the performance of alternative policies. For example, in [1] and [2] a comprehensive model of execution time is used, which attempts to account for the cache hit probability, the concurrency control protocol, and the processing time and queueing delay at hardware resources. These three sub-models are analysed independently and their interactions are captured through a set of non-linear equations. An iterative process solves the system of equations to produce a solution to the overall model. Issues of pipelined execution are not considered.

The focus of the research discussed above is on the benefit of alternatives – hash-join strategies, scheduling mechanisms, cache-coherency policies, etc. – for

handling specific aspects of a DBMS. Models of performance are mainly employed in order to illustrate the benefit of one approach over another and generally there is no attempt to use the developed models to evaluate the performance of actual systems. The models capture the details of particular algorithms or strategy in considerable depth. It is difficult, however, to generalise such models in order to include other aspects that affect the execution of queries in a parallel database.

3.3.2 More comprehensive models

The previous section discussed some studies, in which models of performance were used to prove the benefit of a particular technique or strategy over another. By comparison, there is relatively little work specifically concerned with models of query execution which bring together the various factors affecting performance such as query execution in a pipeline, contention for resources, effect of cache, etc., and are intended for use in the context of actual parallel database systems.

An early attempt in the context of single-processor database systems is the work by Sevcik [83], which proposes an overall framework for predicting resource consumption, throughput and response time, and the way they are affected by various physical and logical database design decisions.

A more recent example is the work by Salza et al [84] (and earlier [85]), in which a modelling methodology for applications running on shared-nothing parallel database systems is developed. A workload model is used to characterise the database relations and set of transactions to be processed. From this, resource utilisation can be estimated. A buffer model is developed to capture the effect of caching. Through bottleneck analysis, maximum system throughput can be predicted. Response time is estimated as follows. First, by using queueing techniques, an estimate is made of how much the execution of each operator is slowed down by the concurrent running of other operators on the same node. A simplifying assumption of exponential service time

distributions is made and thus product-form techniques are applied. Second, the concurrent execution of different operators from the same execution schedule is modelled, taking into account pipelined and partitioned execution. The approach is developed within the context of a particular system (DB2 Parallel Edition on IBM SP2 architecture), but no comparison results are reported.

M. Spiliopoulou et al devote a series of papers [86, 87, 88] to the problem of estimating execution time for queries composed of multiple pipelined operators, scheduled to run in a parallel database system. The intention is to incorporate the execution time prediction mechanism into a generic optimiser for parallel query processing. The developed cost model uses a comprehensive set of analytical formulae to compute the cost of a large variety of query operators, running both in isolation and in a pipeline. Using the formulae, the cost of the query execution plan of the original query can be computed incrementally from the costs of its components. The model does not take into account contention for physical resources.

3.3.3 Commercial tools and products

A number of commercial products now exist, which have performance prediction capabilities. These are usually capacity planning tools, developed for existing parallel database systems. Tools vary in complexity from ones consisting of a simple set of cost formulae to those employing detailed simulation models of the parallel database system.

One example is the DB2 Estimator [89] from IBM, which is an analytical performance estimation tool, designed specifically for DB2 for OS/390 V5 and V6. It runs on a PC and calculates estimated costs using formulae obtained from an analysis of real DB2 code and performance measurements.

SMART (Simulation and Model of Application based Relational Technology) [90, 91] is a tool developed for predicting the performance of relational database

applications using simulation. This tool is currently being supercceded by its re-engineered successor, SWAP. SMART/SWAP is a sophisticated and versatile tool, able to model complex real applications. Currently it models the performance of Oracle7 and is in the process of being extended to model Oracle8.

The Oracle System Sizer V3.0 [92] project at Oracle has, in conjunction with HP and Dell, produced an analytical tool which sizes hardware configurations for Oracle database applications. At present there are two versions, one which predicts configurations for HP NetServers and the other for Dell PowerEdge servers, both running Windows NT. Oracle is planning versions for additional hardware types in the future.

Some other examples of tools are:

- the Athene Performance Management System from Metron Technology [93, 94, 95], an analytical capacity-planning tool, developed for Oracle;
- a simulation-based performance prediction tool for DB2 applications from Platinum Technology [96];
- an analytical tool from BEZ Systems [97, 98] for monitoring and predicting the performance of NCR Teradata and Oracle environments;
- a simulation tool for Oracle from SES Inc [99].

In general, simulation-based tools produce better predictions than analytical tools. Their drawback however is their tendency to be time- and resource- consuming. For both types, there is relatively little published on the quality of the predictions produced by the tools described in this section.

For example, for OLTP applications consisting of lightweight read/write queries, DB2 Estimator aims to predict the resource consumption of at least 90% of the application queries with an error of no more than 10% of measured values. For decision support applications consisting of complex queries that read more than one table and perform joins, the tool aims to predict the performance of at least 80% of the query

types with an error of no more than 20%. However, in some cases cost estimates may be up to 50% higher than measured cost [89]. The analytical tool from Metron aims to predict throughput and resource utilisation to ±10% of measured value and mean response times of queries to within ±15% of measured values together with estimates of variability [93]. However, no published results of the accuracy the tool achieves in practice could be found.

The SMART/SWAP tool is able to predict the throughput of single Oracle7 queries to within 10% of measured values. Even for queries from a complex decision support application predictions for both query throughout and response time are generally better than 15% accurate [124].

Of the various tools mentioned in above, those based on simulation are of less relevance to the concerns of the thesis, while tools based on analytical approaches, such as DB2 Estimator, and the tools from BEZ and Metron, are of more interest. However, only general descriptions of the tools are available in the literature, with very little detail of any of the mechanisms or formulae used.

3.4 STEADY

The analytical tool STEADY [3, 100, 4] has been developed to model a range of different aspects of parallel database system operation in order to compute resource utilisation, system bottleneck(s), and maximum system throughput for a given set of queries (specifying a database application), set of tables, and database configuration. Originally STEADY was developed to model an Ingres Cluster [101] and has been extended to model the Informix XPS and the Oracle7 Parallel Server within this dissertation.

The architecture of the tool is given in Figure 3-3. This served as the basis for development of the new functionality, which includes modules for response time estimation and a cache module. This section provides an introduction to the original

modules of the tool. The extensions developed for STEADY are summarised in Section 6.6, after the underlying models are developed in Chapters 4, 5 and 6.

The input to STEADY consists of four sets of information: a description of the database (relations); a data placement strategy to be used; a DBMS/platform configuration; a database application, represented as the execution plans of SQL queries given as annotated query trees.

Within STEADY there are a number of modules, grouped into three layers. After executing each of these, the system produces as output details of the performance in terms of resource utilisation and maximum throughput. A graphical user interface (GUI) is used to invoke different modules and display their results.

Within the application layer the Profiler is used for generating statistical profiles for the base relations. It is also responsible for estimating profiles for the internal tuple streams that result from the relational operators within the query tree. This is done through selectivity analysis of an operator's predicates, given the profiles of the base relations and/or internal tuple streams on which it operates.

The Data Placement Tool (DPTool) is used to determine how the data within a parallel database is to be distributed. The user selects the strategies to be used and, given these and the application's query trees, DPTool determines how the relations should be fragmented and allocated to different processing elements and discs. A number of declustering and placement strategies are supported [3, 37, 102]. DPTool is primarily used to evaluate various data placement schemes. Users may vary the declustering and placement strategies in order to obtain a combination that leads to the most favourable performance. In addition to this, DPTool estimates the number of accesses (reads and writes) to each fragment from relations involved in the given application.

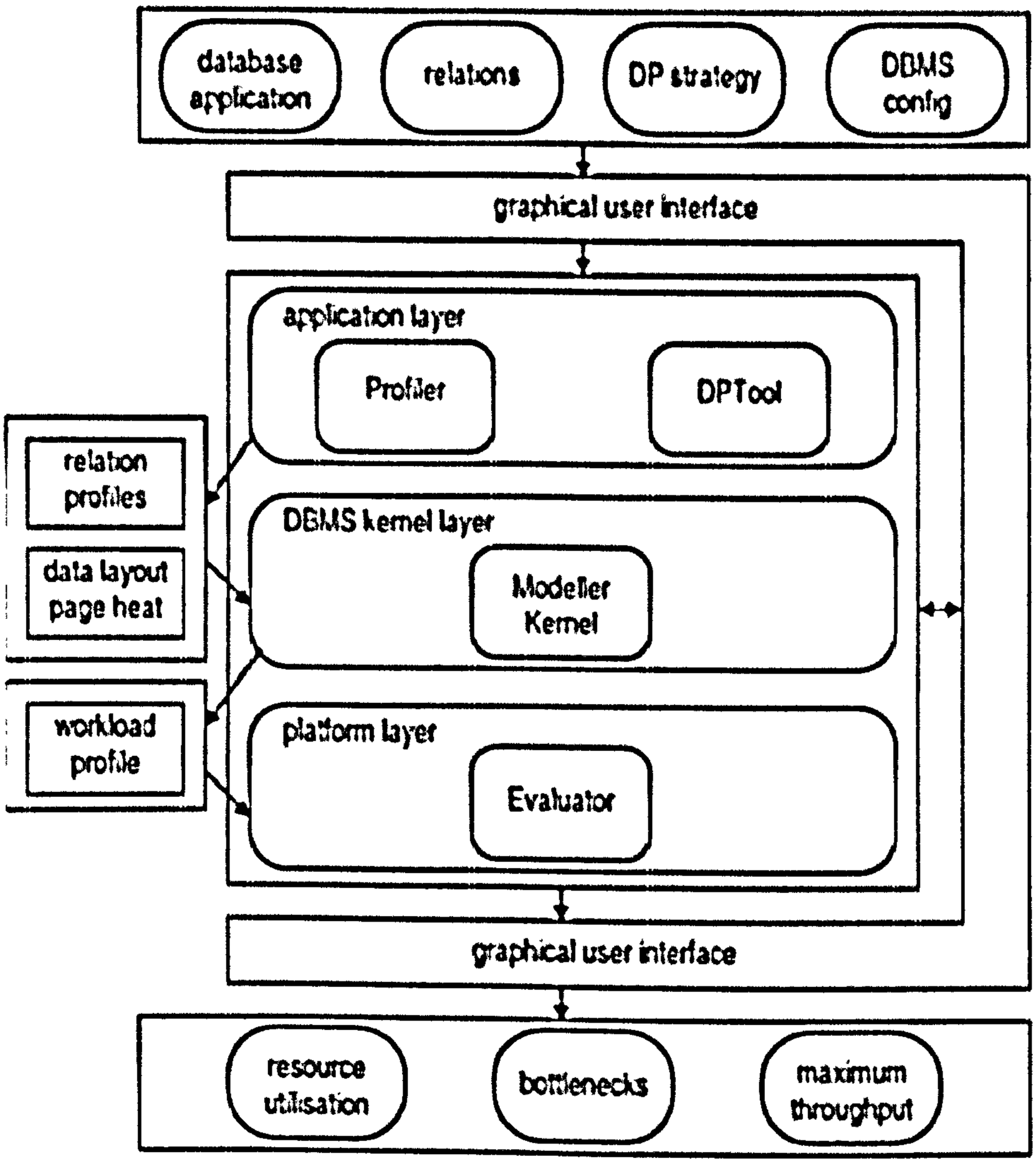


Figure 3-3 Architecture of STEADY

The output of the application layer is a set of relation profiles, a description of the data layout across the discs and nodes and estimations of page access frequencies. The GUI may be used to view and manipulate further the generated data layout.

Within the next layer, the Modeller Kernel produces a profile of the actions the system performs during the execution of the user's application. The profile is expressed in terms of the number of elementary database operations such as *disc_read*, *lock_request*, *predicate_check* etc. performed during execution of the application's transactions. The Modeller Kernel generates the profile based on knowledge of the application's queries, the location of relation fragments (determined by DPTool), and the profiles of tuple streams resulting from relational operators (available from the Profiler). Knowledge of the query execution methods of the database system is used.

The workload profile is generated as output of the DBMS kernel layer and may be viewed through the GUI.

Within the platform layer, the Evaluator maps the workload profile to actual hardware resource consumption. The conversion is based on knowledge of an underlying hardware model of the architecture and of the cost of basic operations. Given the resource consumption, the system bottleneck resource is found. From this, the maximum throughput rate is determined. The maximum transaction throughput gives the user an indication of the upper limit of system capacity in terms of throughput. For this layer, the GUI displays the resource time consumption, the bottleneck resource(s) and the maximum throughput.

3.5 Summary

This chapter has discussed performance modelling and prediction in the context of parallel database systems. Five performance modelling methodologies were reviewed: discrete event simulation, performance Petri nets, performance evaluation process algebra, queueing networks, and the method of layers. Each methodology may be used to obtain performance estimates from a conceptual model of the behaviour of the studied system.

Some representative examples of performance prediction studies in parallel database systems were described. These include studies into specific aspects of parallel database architectures and techniques, as well as more comprehensive models of parallel query execution and the factors that affect it. Different modelling methodologies have been used in these studies, including simple formulae, queueing networks, Petri nets, PEPA, simulation, as well as combinations of the above.

Some existing commercial tools for performance prediction in parallel database systems were reviewed briefly. Typically, these are capacity planning tools that use

analytical models or simulation. The STEADY performance prediction tool, which serves as the starting point if the research carried out in this thesis, was described.

The approach to performance modelling proposed in this dissertation is based on open non-product-form queueing networks. The approach was outlined in Section 1.2 of Chapter 1 and has been chosen in preference to the other methodologies described in this chapter for the following reasons. The computation time and resources needs when using a discrete event simulation model of the system are usually too high. The Petri Nets methodology allows the modelling of a range of behaviours, but the solutions become computationally expensive for more complex models. A similar argument applies to PEPA models. The method of layers proposes a useful framework, but it is not clear whether a MOL model can capture similar degree of detail to that represented in a resource usage profile (discussed in the next chapter). Using product-form queueing networks has the advantage that solutions of the models can be found efficiently, but the type of behaviour that can be represented is restricted.

The approach adopted in this thesis uses a new technique to deal with non-product form queueing network in order to model both the types of synchronisation between stages of query execution and non-exponential service times. It is thus able to take into account contention for resources. The method is based on a representation that captures in considerable detail the patterns of resource consumption by the system. However, the method does not explicitly consider a state space and therefore does not suffer from state-space explosion. Hence, performance estimates can be obtained rapidly. In this respect, the method is practical and has been implemented as the core of the stand-alone STEADY performance prediction tool.

The next chapter discusses the initial stage of the proposed performance prediction method: the construction of the resource usage representation of database activity.

Chapter 4

REPRESENTING DATABASE ACTIVITY AS PATTERNS OF RESOURCE CONSUMPTION

4.1 Introduction

This chapter presents the process of constructing a representation of database activity, from which the sought performance estimates are computed. The representation is constructed from a database application, given as input. The application consists of transactions, which are composed of queries presented as execution plans. This form of the input is discussed in Section 4.2.

The given execution plans are mapped to *parallel execution schedules*. This is represented as a tree structure with nodes representing relational operators (or parts thereof) and links among nodes representing dependencies among operators. Links are also used to represent pipelined execution between two or more relational operators. Such a tree structure can express both inter- and intra- operator parallelism. Execution schedules are the subject of Section 4.3.

The next step is to make explicit the basic operations that make up the relational operators that are at the nodes of the execution schedule tree. The set of basic operations includes constructs such as *obtain_lock*, *read_page*, *and predicate_check*, structured with keywords such as *group*, *option* and *loop*. This is described in Section 4.4.

The final stage is to break down the basic operations themselves into physical resource usage items. The resulting structure is the resource usage block representation of the original application and is described in Section 4.5.

Section 4.6 shows how resource utilisation and maximum throughput may be obtained from the resource usage block profile of a query. Section 4.7 concludes the chapter with a summary.

4.2 Input for the method

The input to the method is a database application, consisting of transactions, each of which is composed of queries. In the context of this thesis, the term application refers to a collection of SQL statements that users may send to a database. It is assumed that this collection can be obtained by extracting SQL statements from a database application.

Within a database application, SQL statements are typically grouped within transactions. Since not all transactions are equally likely to occur, associated with each transaction type is its relative frequency within the application. It is assumed that relative frequencies are either given in advance or established empirically. A different mix of transactions (as in TPC-C, for example) can thus be specified and the effect on performance of varying the frequencies of transaction types can be investigated.

A transaction represents a collection of SQL queries. Queries execute in the order they are given within the transaction. The transaction is considered complete when the last query within it completes. The transactions within an application are the focus of performance estimation, in terms of both overall maximum transaction throughput and mean response time of transactions of each type.

Each of the queries that make up the application's transactions is represented in the form of a query execution plan. This is produced by the query optimiser of the database management system. The automatic generation of a query execution plan from an SQL specification of a query is a topic of considerable research interest [18, 33, 43], but is not of main concern in this study.

The execution plans (also referred to as annotated query trees) of all application queries are assumed to have been generated, and are taken as input. Two example queries and their execution plans, presented in the notation adopted for the STEADY tool, are given in Figure 4-1.

```
SELECT max(int)
FROM un270k
WHERE key > 0

AGGREGATE( SELECT( SCAN, un270k, ATTR(un270k, 1) > 0),
           MAX(ATTR(un270k, 2)))
```

(a)

```
SELECT max(un30k.int),
       count(un30k.int)
FROM un60k, un30k, un50
WHERE un60k.key = un30k.key AND
      un30k.key = un50.key

1   AGGREGATE( JOIN( HASH,
2                    JOIN( HASH,
3                          SELECT(SCAN, un50, TRUE),
4                          SELECT(SCAN, un60k, TRUE),
5                          ATTR(un50, 1) = ATTR(un60k, 1),
6                          ATTR(un50, 1)),
7                    SELECT(SCAN, un30k, TRUE),
8                    ATTR(un30k, 1) = ATTR(un50, 1),
9                    ATTR(un30k, 2)),
10          (MAX(ATTR(un30k, 2)), COUNT(ATTR(un30k, 2))))
```

(b)

Figure 4-1 Two example query execution plans

Plan (a) corresponds to a query that finds the tuple(s) with the largest value for attribute *int* from a subset of the tuples of relation *un270k*. This is an AS^{3}AP *Uniques* relation (see Section 2.4.2) with 270,000 tuples. The subset is chosen to contain all tuples with a positive value for attribute *key*.

Initially, tuples from relation *un270k* that have a positive value for the first attribute are chosen. The SELECT operator performs this operation. Its first argument (SCAN) signifies the access method that has been chosen by the query optimiser to read the tuples from disc. The value SCAN denotes a full table scan. An alternative could

have been BTREE, which signifies the use of a btree index for accessing the relation's tuples. The set of tuples, formed by the SELECT operator is provided as input to an AGGREGATE operator, which computes a function of all the tuples from the set it takes as its first argument. In this case, the MAX function is computed across the second attribute, which is *int*. The tuples with the largest value are returned as the result of the query.

The execution plan given in Figure 4-1(b) represents a query joining three relations. The query joins three *Uniques* relations *un60k*, *un30k*, and *un80*. Three tuples (one from each table) are joined if they have the same value for the *key* attribute. The resulting *un30k* tuple(s) with the largest *int* value are returned to the user. A count of all such tuples is also returned.

In lines 3 and 4 relations *un80* and *un60k* are scanned and joined using a hash-based join method (line 2). There is no condition attached to the selection, indicated by the keyword TRUE. The join condition is equality between the first attributes of the two relations (line 5), which are *key*. Since no attributes from either relation are further required for the query, the tuple stream resulting from this join contains only values of the *key* attribute (line 6). This tuple stream is then joined to the *un30k* relation, which is scanned in line 7. Again, equality on the *key* attributes (line 8) is the join condition. The second attribute of the resulting stream (*int*) is returned in line 9 to the final operator which performs MAX and COUNT (line 10) aggregation on the *int* attribute.

4.3 Execution schedule

The next step is to transform the execution plan into an operator tree, created from a query execution plan by breaking down each of its nodes into one or more constituent operator nodes and identifying dependencies among nodes. Some of the nodes of the annotated query tree (i.e. the execution plan), e.g. SCAN, are mapped directly to operator tree nodes. Others, such as JOIN, are decomposed further into

constituent nodes. In the case of a hash-based join two operator nodes – *build* and *probe* – are created and a dependency between them is established. A mapping of system processors to operators of the tree constitutes a *parallel execution schedule*.

An example parallel execution schedule is presented in Figure 4-2. The schedule is for the query whose execution plan is shown in Figure 4-1(a). The schedule consists of operator nodes, referred to as blocks, each of which has a *name*, a *mode* and a *home*. The name is used to identify the block. There are blocks for initiating and finishing the execution plan (blocks start and end, respectively), a block for the SELECT operator and two blocks for the AGGREGATE operator.

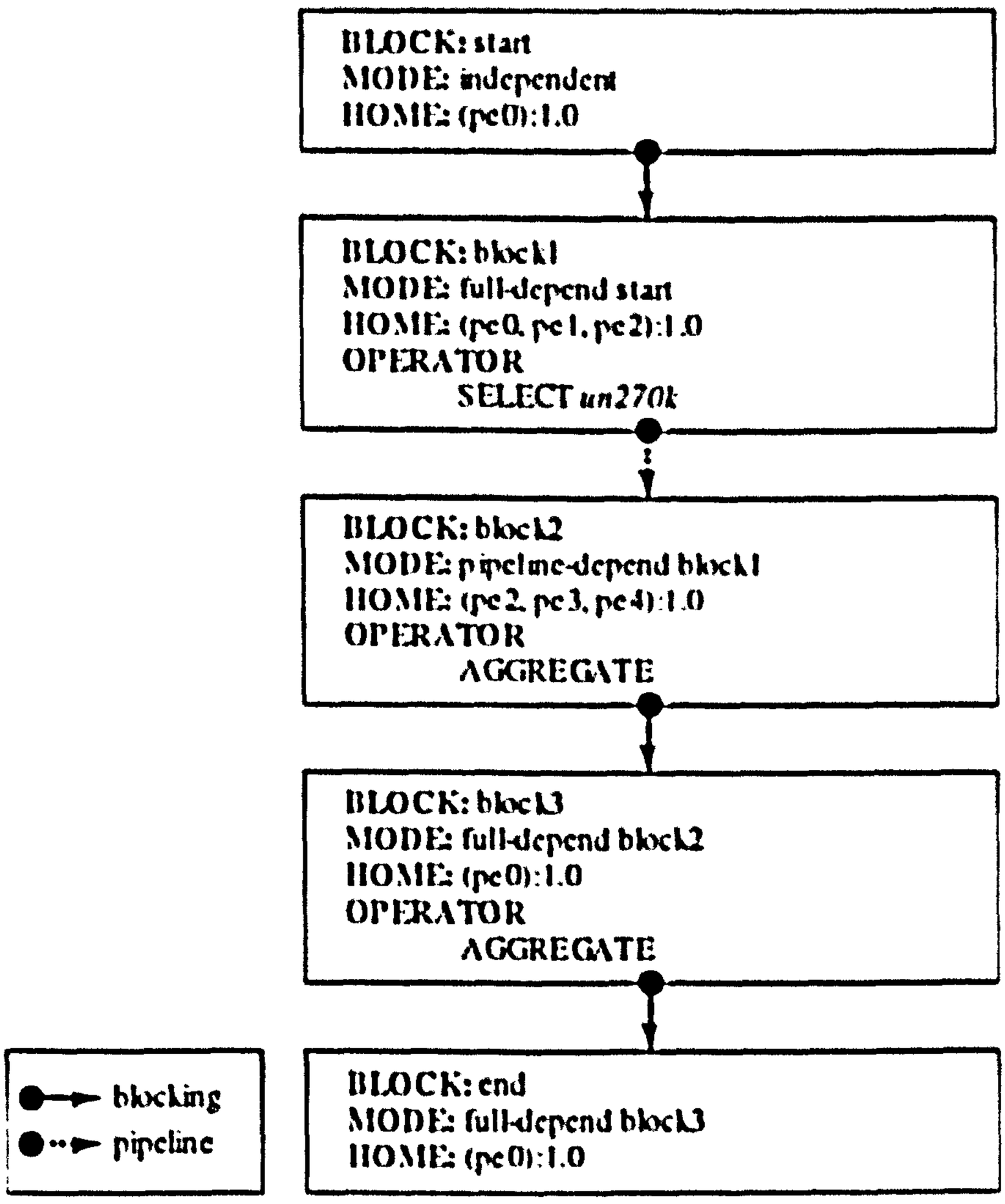


Figure 4-2 Execution schedule

The modes of blocks represent flow of data as well as timing dependencies among blocks. A block may have *pipeline* dependency or *full* dependency (blocking) with one or more other blocks. The block that initiates the schedule has *independent*

mode. To emphasise block dependencies, arrows are drawn in Figure 4-2. Dotted arrows denote pipelined execution. For example, the scanning of tuples from *um270k* in block 1 and their aggregation in block 2 can be done in a pipeline manner. Solid arrows denote blocking. Thus block 1 can begin after block start has run. Similarly, block 3 runs only after block 2 has performed aggregation. Note that the original aggregation operator is performed within two blocks, reflecting the 2-stage approach adopted by Informix XPS. Informix partitions the stream of scanned tuples to a number of co-servers on separate PEs, which carry out aggregation in parallel. Upon completion, each aggregate thread sends its computed aggregate to one co-server (the one that originally initiates the query) for a final aggregation.

The home of a block is used to specify partitioned parallelism. The operator of the block may be spread across multiple PEs, as specified in the figure. For example, the scanning of *um270k* is carried out on PEs 0-2, as dictated by data placement constraints. Similarly, after redistributing the tuples selected from *um270k*, the first stage of the aggregation takes place across PEs 2-4. When the same PE participates in two stages of the same pipeline it has to alternate between the two tasks it is involved in.

A more complicated schedule, corresponding to the execution plan in Figure 4-1(b), is given in Figure 4-3. The schedule reflects a left-deep operator tree (see Section 2.3.4). The homes of blocks are not shown.

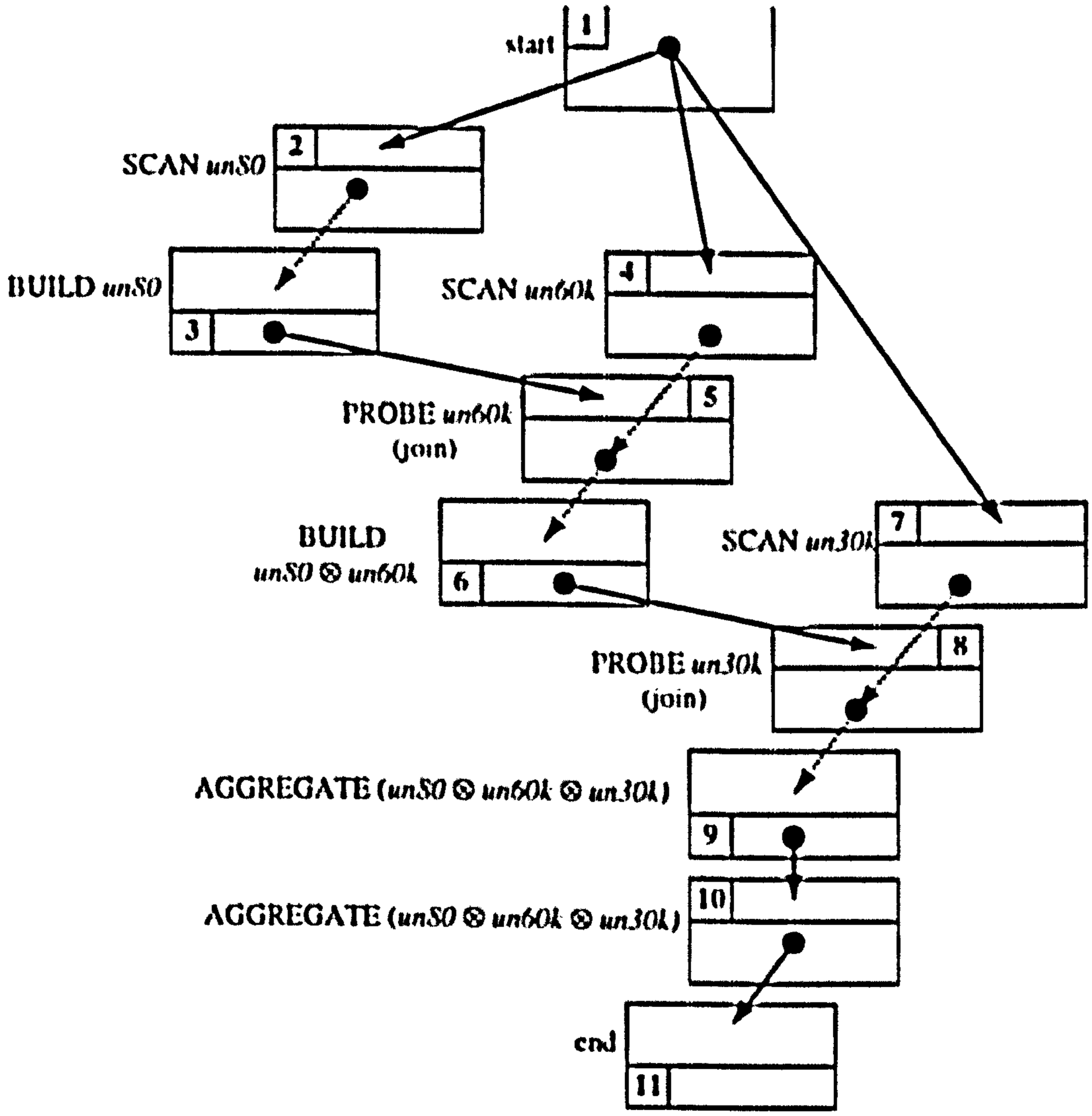


Figure 4-3 More complex execution schedule

The scanning blocks (blocks 2, 4, and 7) can begin reading database pages after an activation signal has been received from block 1. Moreover, blocks 4 and 7 are indirectly dependent upon the completion of other blocks. For example, the scanning of relation *un60k* in block 4 is coupled with the probing of the hash table in block 5, which can take place only after the hash table has been constructed in block 3. Only after the hash table is fully built, it can begin to be probed. The full dependency between blocks 3 and 5 captures this constraint. Thus, the execution sequence for the first part of this three-table join query is:

1. block 1;
2. block 2 coupled with block 3;
3. block 4 coupled with block 5.

The tuple stream resulting from the joining of relations *un80* and *un60k* in block 5 is used to build a second hash table in block 6 in preparation for the join with the third relation, *un30k*. This establishes a three-stage pipeline involving blocks 4, 5 and 6. This pipeline can be scheduled only after the creation of the hash table for relation *un80* has completed in block 3, as required by the full dependency between blocks 3 and 5.

Once the hash table for *un80* ⊙ *un60k* (the result of joining relations *un80* and *un60k*) has been created, the scanning of relation *un30k* can begin. As with the first part of the query, the scanning of *un30k* in block 7 is coupled with the probing of the hash table for *un80* ⊙ *un60k* in block 8.

Resulting tuples from block 8 are passed on down the pipeline to block 9, which performs MAX aggregation. Another three-stage pipeline is thus established, involving blocks 7, 8 and 9. As in the previous example two blocks – 9 and 10 – are carrying out MAX aggregation. The last block in the sequence, block 11, represents the result of returning the overall maximum to the user.

4.4 Task blocks

In the next level of the representation each operator in the block is decomposed into lower-level basic operations, reflecting the sequence of actions taken by the system when executing the operators. This is achieved through a 'macro expansion' of the operators in the blocks, while the rest of the block structure remains unchanged. Such expanded blocks form the *task block representation* of the query. This is illustrated for the SELECT block (block 1) from Figure 4-2.

The task block is shown in Figure 4-4 (a). Represented is the process of consecutive fetching of database pages of *un270k* from disc, selecting relevant tuples, and sending them on to another block. Two types of notation are used in the body of the block (lines 5-18). Basic operations such as obtaining a lock (line 6), reading a page (lines 7-8), checking whether a tuple satisfies a predicate (line 10) and sending a tuple

(lines 12-15), are given in italics. The read operation indicates the number of pages to be read and disc to be used (1 and *disc0*, respectively) by each PE from the block's home. Associated with the read operation is the cache miss probability. The estimation of this probability (0.00, 0.76 and 0.98 for PE0, PE1 and PE2 in this case) is the subject of Chapter 8. It is used to indicate that a disc read is not required for a page already in cache.

```
1   BLOCK: block1
2   MODE: full-depend START
3   HOME: (PE0, PE1, PE2)
4   OPERATION DEFINITION

5    loop (PE0:4528; PE1:4740; PE2:4232){
6     obtain_lock;
7     read (PE0.disc0(1), PE1.disc0(1), PE2.disc0(1))
8      PE0 0.00 & PE1 0.76 & PE2 0.98;
9     loop (20) {
10     predicate_check;
11     group {
12      send block2 155
13         PE0-(PE2 0.33; PE3 0.33; PE4 0.33);
14         PE1-(PE2 0.33; PE3 0.33; PE4 0.33);
15         PE2-(PE2 0.33; PE3 0.33; PE4 0.33)
16     } 1.0
17    }
18   }
    END DEFINITION
```

(a)

```
1   BLOCK: block2
2   MODE: pipeline-depend BLK1
3   HOME: (PE2, PE3, PE4)
4   OPERATION DEFINITION

5    loop (PE2: 90000, PE3: 90000; PE4 90000){
6     receive 1,
7     aggregate;
8    }
9    send block3 4
10     PE2-(PE0 1.0);
11     PE3-(PE0 1 0);
12     PE4-(PE0 1 0)

    END DEFINITION
```

(b)

Figure 4-4 Example task blocks

The second type of notation represents templates used to structure basic operations. Templates are indicated in bold. Two types of template are used in this block: group and loop. The group template is used to designate a sequence of one or more operations, which has an associated probability of occurring. For example, a group template in lines 11-16 encloses the send operation. The associated probability is used to restrict the number of tuples that are sent to the next block. Since in this case all tuples satisfy the original query predicate (ATTR (un270k, 1) > 0 in Figure 4-1), all tuples are sent and the group probability is 1.0. The probability is estimated through predicate selectivity analysis by the Modeller Kernel and the Profiler of STEADY (see Section 3.4).

The loop template is used to designate repetition. The consecutive fetching of data pages is conveniently expressed with the loop construct since similar actions are carried out for each fetched page. The loop on line 5 represents the processing of 4528, 4740 and 4232 pages from PE0, PE1, and PE2, respectively, which reflects the fragmentation of the *wi270k* table.

For each page read, a second loop (line 9) is used to represent the processing of individual tuples from the newly read page. After checking whether the predicate holds (line 10) each of the 20 tuples within the page is sent to block 2. The tuple size is 155 bytes (line 12). The details of tuple redistribution are specified in lines 13-15. In this case, each PE from the home of the block distributes its tuples evenly among the PEs from the home of the receiving block (block 2). This reflects the Informix XPS redistribution, which is carried out by the exchange operator (see Section 2.5.2).

Another task block (block 2 from the schedule in Figure 4-2) is shown in Figure 4-4(b). The block begins with a loop, which receives all 270,000 tuples sent by block 1. The tuples are evenly distributed between the three PEs of the block's home. After receiving a tuple, an aggregation operation is performed on it. For this query, this involves comparing the received tuple with the current maximum and swapping values if necessary.

The two blocks are linked with pipeline dependency. This means that the fetching of pages, checking predicates, and sending of tuples (in block 1) and their aggregating (in block 2) are all done in a pipeline. Once the aggregation is complete, the computed maximum value from each PE is sent to the next block (block 3) where the final aggregation is computed.

4.5 Resource blocks

In the final stage of the representation, a second macro expansion is applied to each of the basic operations of the task blocks. This time, each basic operation is

transformed into a detailed sequence of resource consumption. The set of resource usage blocks and their interdependencies (taken from the execution schedule) form the *resource usage profile* of the query. For example, consider Figure 4-5, which shows the resource usage block corresponding to the task block discussed in Figure 4-4(a). The processing of pages and tuples is now represented as a detailed sequence of resource consumption.

```
1     BLOCK: BLK1
2     MODE: full-depend START
3     HOME: (PE0, PE1, PE2)
4     RESOURCE TIME

5      loop (PE0:4528,PE1:4740,PE2:4232) {
6       option {
7        pu to: 0.6
8        pu to: 0.4;
9       };
10      group {
11       pu t1;
12       disc0 (PE0(t5), PE1(t6), PE3(t7));
13      } PE0:0.00,PE1:0.76;PE2:0.98;
14      loop (PE0:20,PE1:20,PE2:20) {
15       pu t4;
16       group {
17        group {
18         pu t10;
19         ssu t12;
20         net t9
21        }PE0:0.67;PE1:0.67;PE2:0.67
22       }1.0
23      }
24     }
       END TIME
```

Figure 4-5 An example of a resource usage block

The usage of a particular hardware resource (e.g. PU, SSU, net and disc) for a specified amount of time is represented in italics. Simple examples are lines 11, 15,18-20 where a resource is used for time t_i. Line 12 gives an example of a resource (disc0) used for different amounts of time depending on the processing element (PEs 0-2) using the resource. The service time requirements for simple resource usage (t_i) are extracted from the particular parallel database architecture modelled. For example, t_5, t_6, and t_7 in line 12 represent the average time spent by disc0 of PE0, PE1, and PE3 in the process of

fetching a data page, while t_0 (line 11) is the amount of service time consumed by the PU resource during the page fetch.

The same templates used in the task blocks are also employed in the resource blocks. The group template is used to designate a sequence of resource usage items, which has an associated probability of occurring. For example, the group template in lines 17-21 encloses three simple resource items, which represent the resource consumption required for the sending (see lines 11-16 in Figure 4-4(a)) of a selected tuple for processing by block 2. The probability associated with this group template is intended to rule out the physical sending across the interconnecting network of tuples destined for the processor where they are produced. Similarly, the group template in lines 10-13 is used to incorporate the cache miss probability into the resource usage block.

An option template is also used in the block. It is used to denote choice. It can have a number of branches, each of which has an associated probability and contains a sequence of resource usage items. The grouped items occur together with the given probability. An example of option template use is given in lines 6-9, where the PU resource is used for either time t_9 (with probability 0.6) or time t_{10} (with probability 0.4).

4.6 Calculating maximum throughput

The developed representation can be used to estimate resource utilisation and maximum system throughput. This can be done in the following three steps:

1. Compute the utilisation of each resource by traversing the resource blocks. For example, consider the *pu* resource of PE0 in a transaction, represented by the single resource block from Figure 4-5. Suppose the transaction arrival rate is λ. Given the visits by the transaction to the *pu* resource, its utilisation is computed as:

$$\rho(pu) = \lambda(4528 \times 0.6 \times t_9 + 4528 \times 0.4 \times t_{10} + 4528 \times 0.0 \times t_1 + 4528 \times 20 \times t_8 + 4528 \times 20 \times 0.67 \times 1.0 \times t_{11})$$

This is, in effect, an application of formula (3.1) from Section 3.2.4.1 for computing the utilisation of a resource within an open queueing network. Note that there is no need to solve a system of equations in order to obtain the visit ratios explicitly, since their values can be computed easily from the elements of the resource usage profiles, i.e. the *loop*, *group*, and *option* templates.

2. Compute the maximum possible arrival rate each resource can handle by equating its utilisation to 1. For the *pu*, this is:

$$\lambda_{max}(pu) = \frac{1}{D(pu)}$$

3. Find the resource or set of resources with the lowest value of λ_{max} from those computed in step 2. This resource is the bottleneck for the given application. It is the first resource to become saturated as the transaction arrival rate increases and the highest rate it can handle represents the maximum system throughput.

4.7 Summary

This chapter has presented the process of constructing the resource usage representation of an executing query. A database application with transactions composed of queries, which are given as execution plans, is first transformed into a parallel execution schedule. The transformation is based on thorough understanding of the query execution mechanisms of the modelled database system. The execution schedule is then mapped to a series of task blocks, each of which expands to a corresponding resource block. The resource block representation of a query captures in detail the sequence of resource consumption, necessary to execute the query. Maximum system throughput may be computed directly from the resource usage representation, while mean response time can be obtained through further analysis.

The resource block representation has been introduced with the help of concrete examples in order to aid understanding. However, the method can be applied in

general, as shown in Chapter 7, where a range of queries, representative of those found in basic decision support applications, are analysed.

The task block/resource block notation can be used to represent other aspects of database activity, not directly responsible for query execution. For example, by associating probabilities with basic costs within the task block notation, the effects of caching may be accounted for (see Section 4.4). Other types of background activity, such as the flushing of database buffers that occurs during periodic checkpoints, may be modelled with simple task blocks consisting of appropriate basic cost items.

The next two chapters discuss at length the process of computing transaction response time from its resource usage representation.

Chapter 5

ESTIMATING RESOURCE WAITING TIME

5.1 Introduction

In the second stage of the performance estimation method the aim is to estimate the response time of individual hardware resources, given their pattern of use as specified by the resource usage profile. The resource usage profile is mapped to an open queueing network. A heuristic rule is devised to assign an M/G/1 or M/M/1 label to each resource. In accordance with the assigned label, standard formulae from queueing theory can be applied to calculate waiting time for hardware resources and, from these, the response time of the resource usage profile.

Section 5.2 introduces briefly the M/M/1 and M/G/1 queueing systems, which are used in later sections of the chapter. Section 5.3 discusses how resource usage profiles can be mapped to queuing networks. Two examples of resource usage profiles and corresponding queueing networks are presented to illustrate the process. The difficulties with such networks are explained. In Section 5.4 some approximation techniques for open networks with non-exponential service times are introduced and applied to the two examples from Section 5.3. Section 5.5 applies the techniques to a much larger set of examples in order to assess their suitability. Section 5.6 presents the heuristic rule and results obtained using it. Instead of reporting individual resource waiting times, the results presented are for the overall response time of the resource usage, which is computed from the estimated values of these quantities. Section 5.7 summarises the chapter.

5.2 The M/M/1 and M/G/1 queues

This section introduces the M/M/1 and M/G/1 queues, which are used throughout this chapter. In addition, the concept of *squared coefficient of variation* is introduced in relation to the service time process of an M/G/1 queue.

The M/M/1 queue is a queueing system with exponentially distributed inter-arrival times, exponentially distributed service times and a single server. The letter 'M' stands for 'Markovian' and is used to denote the exponential arrival and service processes. The two processes are characterised by the average arrival rate λ, and the mean service time $\bar{x}$, respectively. Performance measures for an M/M/1 system are easily computed. If $\rho = \lambda\bar{x}$ denotes the utilisation of the queue, then the average number of customers in the system is give by

$$N = \frac{\rho}{1-\rho} \tag{5.1}$$

and the average delay by

$$T = \frac{\bar{x}}{1-\rho}$$

The M/G/1 queue has an exponential arrival process and a single server whose service time has some general distribution independent of the arrival process. The arrival process is characterised by its rate λ, while the random variable describing the service process is characterised by its mean (first moment) $\bar{x}$ and its second moment $\overline{x^2}$. The first two moments are sufficient to derive performance measures for the system, but instead of working explicitly with them, it is more convenient to use the squared coefficient of variation of the service process. This quantity is defined as:

$$Cs^2 = \frac{\overline{x^2}}{(\bar{x})^2} - 1$$

The celebrated Khinchin-Pollaczek formula [103] can be used to calculate performance measures for the M/G/1 queue. For example, the average number of customers in the system is given by

$$N = \rho + \frac{\rho^2(Cs^2 + 1)}{2(1-\rho)}$$

Note that the M/M/1 queue is a special case of the M/G/1 queue. The exponential distribution has a squared coefficient of variation equal to 1. Substituting 1 for Cs^2 in the Khinchin-Pollaczek formula reduces it to the simple formula (5.1) above for M/M/1 queue. The Khinchin-Pollaczek formula also shows that the number of customers in the system grows with the squared coefficient of variation. Thus an M/G/1 queue with squared coefficient of variation of service time greater than 1 has a longer queue than an M/M/1 queue. This is exploited by the heuristic proposed in Section 5.6.

Section 3.2.4.1 discussed product-form networks. Networks composed of M/M/1 queues are in product form. However, those composed of M/G/1 queues are in general not in product form. Such networks must be treated as composed of G/G/1 queues, i.e. queues with general distributions for both the arrival and service processes. Unfortunately, no exact analytical techniques exist for solving such networks.

5.3 Using queueing networks

The resource usage profile of a query is taken to specify a queueing network, which can subsequently be "solved" for resource response time. The queuing stations of the network are the hardware resources of the machine (e.g. PU, Deltanet, discs, etc.) and are treated as FCFS servers. This is a reasonable assumption for the communication hardware and the discs. However, threads requesting service at the PU are likely to be serviced according to a more complex discipline and the FCFS discipline is assumed for convenience. The transitions between stations are determined by the templates within the resource usage blocks. This is illustrated with two examples.

The first example involves a simple application consisting of two single-query transactions, T_1 and T_2, as shown in Figure 5-1. Transaction inter-arrival times are assumed to be distributed exponentially with rate λ; the probability that a newly arrived transaction is of type T_1 (respectively T_2) is 0.4 (respectively 0.6). The patterns of resource consumption for the two hypothetical queries are shown. In the case of T_1, resources *resA*, *resB* and *resC* are repeatedly utilised for the specified constant amount of time; similarly for T_2.

This pattern of resource consumption can be represented as an open queueing network with two customer classes T_1 (solid line) and T_2 (dashed line), as illustrated. Shown are the three resources, the probabilities of transitions among them, and the service times for each customer class and resource.

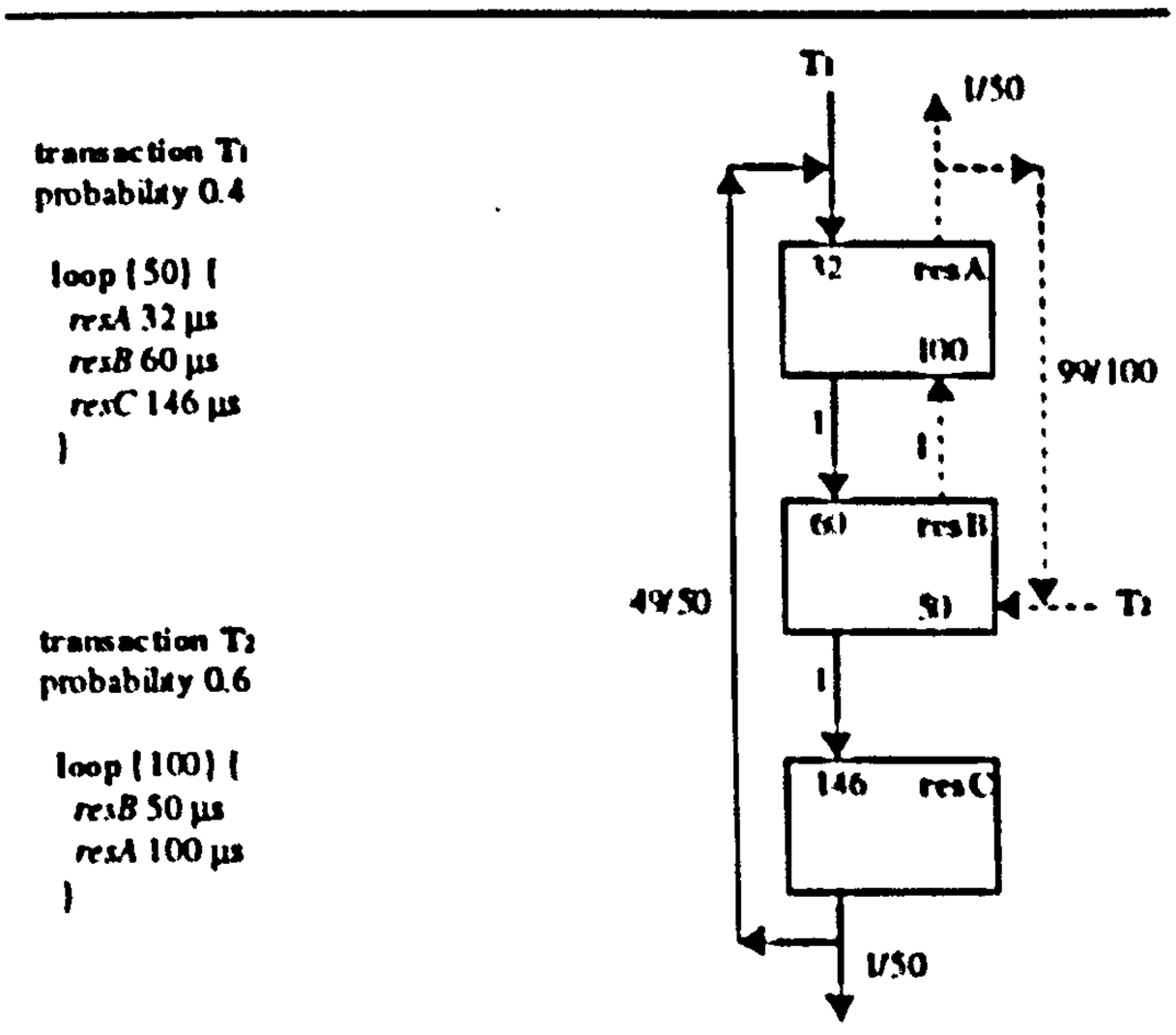


Figure 5-1 Example of queueing network

The second example represents a more concrete situation. Figure 5-2 shows the patterns of resource usage of a query representing the fetching of database pages. The query begins by sending 4 network packets within a loop. After using the *ssu* and *pu* resources for 80 and 56 µs respectively, it enters a loop to fetch 186 database pages. A page may or may not already be in the database cache. A group template is used to denote the operations to be carried out in case the requested page is not to be found in

the cache. Associated with this is the cache miss probability (0.9475). In the case of a cache hit, only the resource consumption following the group will take place. On the other hand, a page that needs to be fetched from disc may be located on any of the four discs. An option template is used, with probabilities of (0.25 in this case) to specify the likelihood of a particular disc being involved. Since a page may be located anywhere on the disc, the disc service time is a combination of the seek time (s, $0 \le s \le 17$ ms), rotational delay (rd, $0 \le rd \le 8.33$ ms) and block transfer time (btt = 0.66 ms). Once a page has been fetched or found to be in the cache, certain tuples from it are chosen and sent as packets via the network to some other node. This is represented by the usage of the last three resources in the loop. The corresponding queueing network is also shown.

```
transaction T
probability 1.0

loop (4) {
 pu 32 µs;
 ssu 60 µs;
 net 146 µs
}
ssu 80 µs;
pu 56 µs;
loop (186) {
 pu 40 µs;
 group {
  pu 2551 µs;
  ssu 40 µs;
  option {
   disc0 (s + rd + btt) ms : 0.25
   disc1 (s + rd + btt) ms : 0.25
   disc2 (s + rd + btt) ms : 0.25
   disc3 (s + rd + btt) ms : 0.25
  }
 } 0.9475
 pu 181 µs;
 ssu 340 µs;
 net 827 µs;
}
```

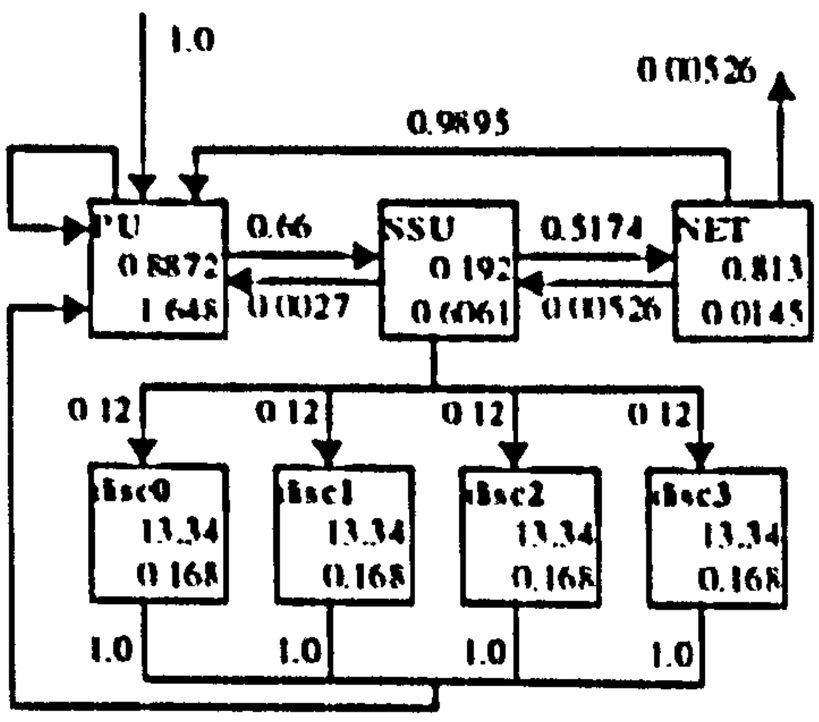


Figure 5-2 A more concrete transaction and corresponding queueing network

The two numbers in each box are the mean service times in milliseconds (first number) and squared coefficient of variation at each queue (second number). Both can be computed from the resource block representation (or the network). Consider for example the *pu* resource. Within a single transaction it is used 4 times for 32 µsec, once for 56 µsec, 186 times for 40 µsec, $186 \times 0.9475 = 176.2$ times for 2551 µsec, and 186 times for 181 µsec. Its mean service time is therefore

$$\overline{x_{pw}} = \frac{4\times32+1\times56+186\times40+186\times0.9475\times2551+186\times181}{4+1+186+186\times0.9475+186} = 887.2\,\mu\text{sec},$$

while its second moment is

$$\overline{x_{pw}^2} = \frac{4\times32^2+1\times56^2+186\times40^2+186\times0.9475\times2551^2+186\times181^2}{4+1+186+186\times0.9475+186}$$
$$= 2084585.104.$$

From these two quantities, the squared coefficient of variation can be computed:

$$Cs_{pw}^2 = \frac{\overline{x_{pw}^2}}{(x_{pw})^2} - 1 = \frac{2084585.104}{(887.2)^2} - 1 = 1.648$$

Two difficulties present themselves when working with queueing networks in this context. One concerns the service time requirements at the queueing stations. Typically, exponentially distributed service times with a given mean are assumed. Thus, the network is taken to be composed of M/M/1 servers. However, in practice assuming non-exponential service times is more realistic. The queueing network is thus composed of non-exponential servers, which renders it a non-product-form queueing network.

There is a second reason why the networks are not in product form, namely the dependencies among resource usage blocks (e.g. pipeline execution and blocking). This imposes a synchronisation mechanism on the queueing network, which governs the timing of transitions between stations. The synchronisation is not captured with these simple examples, but reflects the execution of complex queries as specified by the (original) parallel execution schedule.

There have been numerous studies that address the non-exponential service time problem and lead to analytical approximation techniques [63]; similarly, there are studies, concerned with synchronisation behaviour, such as occurs in fork-join systems [68]. The approach used here is to tackle the two problems separately. In this chapter the first problem is addressed without regard for resource block inter-dependencies (i.e. the transitions between queueing centres are assumed to be of the "classical" type); the

synchronisation issue is dealt with in the following chapter where overall query response time is estimated from the resource usage profile.

5.4 Approximation techniques and their application

The non-product-form nature of the networks, due to non-exponential service time requirements, means that no exact analytical solutions for the response time or queue length of queueing centres can be used. However, numerous approximation techniques for non-product-form networks have been proposed in the literature. In [63] a chapter is devoted to algorithms for dealing with non-product-form queueing networks. A number of approximation techniques are applicable, depending on the type of non-product-form network: open, closed, or mixed. Based on the listed methods for open networks, approximations to the waiting time at queueing centres were sought as follows:

- **Treat all resources as M/G/1 queues.** In this simple-minded approach the mean queue length at each resource (including the request being serviced) is estimated using the Khinchin-Pollaczek [103] formula for an M/G/1 queue:

$$QL_i = \rho_i + \frac{\rho_i^2(Cs_i^2 + 1)}{2(1 - \rho_i)}$$

Here Cs_i^2 is the squared coefficient of variation of service time at queueing centre i and ρ_i is its utilisation. From this the mean waiting time at each queue and the mean residence time of a customer is easily derived. Note that in order to apply the formula the value of ρ_i is obtained from $\rho_i = \lambda_i x_i$, where x_i is the mean service time at queue i and λ_i is its throughput. The throughput can be calculated from the traffic rate equations of the network (see Section 3.2.4). Note that even though the formula correctly accounts for the service time distribution through Cs_i^2, the distribution of the arrival process is not exponential in general. Thus regarding all resources as M/G/1 queues is only an approximation.

- **Use a maximum entropy approximation.** In [104, 105, 106] the principle of *maximum entropy* is applied to analyse open queueing network models with FCFS discipline, multiple customer classes, and general service pattern at each queue. The principle of maximum entropy states that of all probability distributions that may be used to characterise the (unknown) state of the system, the minimally prejudiced one should be chosen. This distribution is the one that maximises the entropy function of the system and can be found from the available information about the system. An implication of the use of the maximum entropy formalism is that queueing centres can be analysed in isolation, once the flow at each queueing centre is determined. This can be done by using a generalised exponential distribution to approximate the arrival, service and departure processes of each queue in the network. The full details of the method are given in [104].
- **Use a decomposition method.** In [62] Gelenbe and Pujolle describe an approximation method for estimating the mean number of customers at each queue of the network. The method allows each queue to be analysed independently from the rest of the network. This is based on the assumption that the departure process from any station is a renewal process: the time interval between two departures does not depend on the preceding intervals. The full details of the method are given in [62]. Several variations of this method have been proposed, notably by Kuehn [107] (not applicable to multi-class networks), Pujolle [108], Whitt [109, 110], and others. The Gelenbe and Pujolle method was selected as representative of this class of approximation techniques. It also appears to be the most recent.
- **Use mean value analysis.** A mean value analysis method is proposed in [62] for the estimation of number of customers in a queue for the case of open networks with general service time requirements. The basic idea is that the mean response time of a station of a general network is the sum of the mean service times and the mean waiting times before being served. This last quantity depends on the length of the

queue at the instant of arrival. For product-form networks, the length of the queue at the arrival instant is the same as the mean number of customers in the stationary case. This property is not true for non-product-form networks, but the authors nevertheless assume it holds and thus obtain an approximation to the mean response time.

Two more techniques were considered, but not used. Diffusion approximation techniques were not investigated since at present no solutions are available for multiple class networks [63]. Also, the method by Harrison and Nguyen [111], which uses reflected Brownian motion to approximate the queue length of an open queueing network, was not used, since the tool implementing the method was not able to run with the networks used here.

Each of the approximation techniques listed above has been applied to the examples for different transaction arrival rates, (resulting in corresponding bottleneck resource utilisations) and the results produced are compared against results obtained by simulation. The CSIM18 Simulation Engine [112] from Mesquite Software was used to perform the simulations. Results were obtained using a 95% confidence level and two decimal places of accuracy.

Rather than comparing individual resource queueing times, the overall transaction response time (computed from the estimated values of these quantities) are compared. Figures 5-3 and 5-4 show the mean response time for customers of type T_1 and T_2 for the transactions in Figure 5-1.

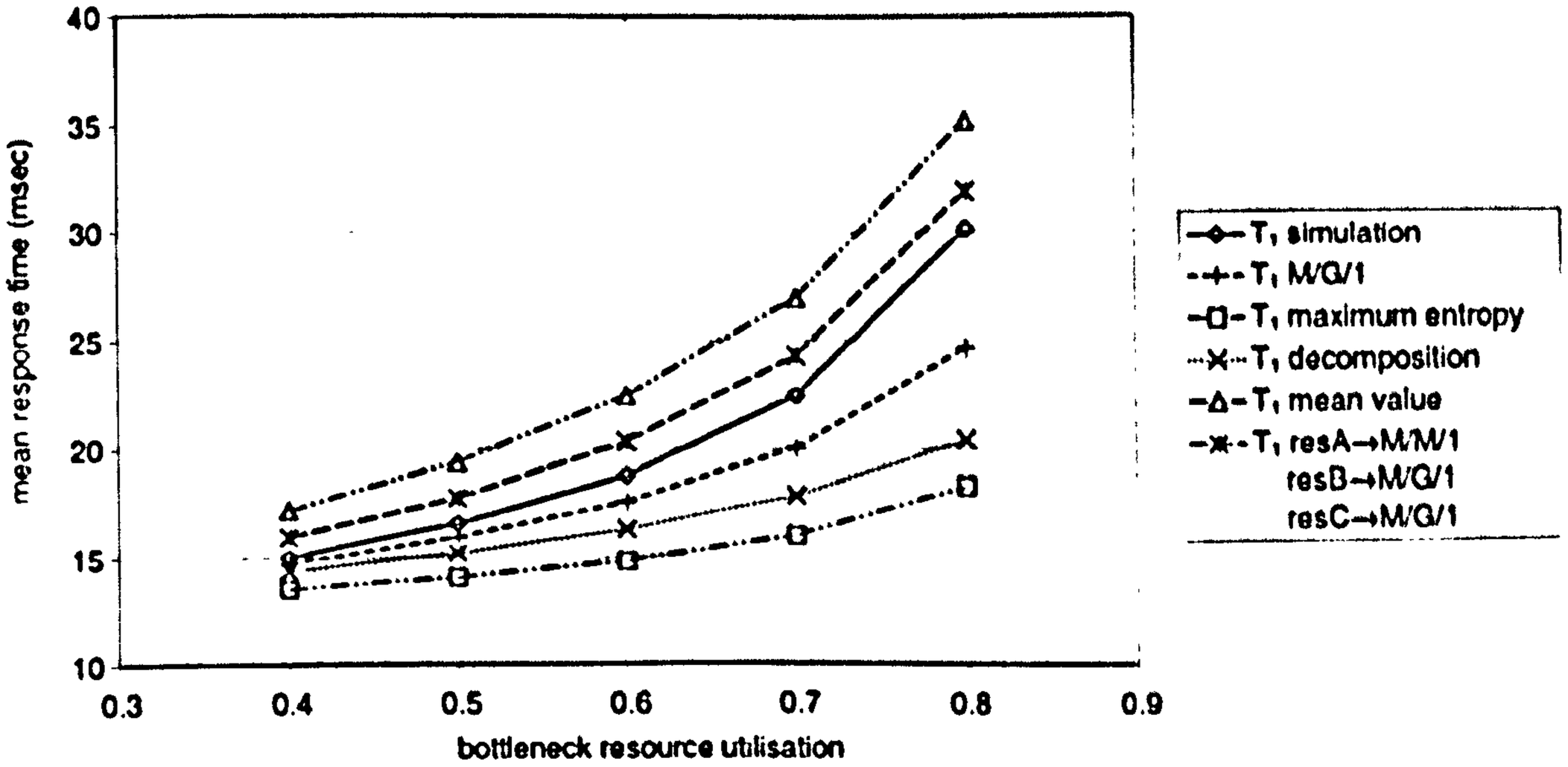


Figure 5-3 Mean response time of T_1 transactions

The *y*-axis gives the response time in milliseconds, while the *x*-axis measures the utilisation of the bottleneck resource of the network (resource *resA* in this case). The required degree of utilisation is achieved by setting appropriately the overall transaction arrival rate parameter λ. In addition to the predictions of the methods listed above, the mean response time computed by treating queue *resA* as an M/M/1 resource and queues *resB* and *resC* as M/G/1 resources is also shown. This combination provides the best match to the simulation results when compared with the other methods.

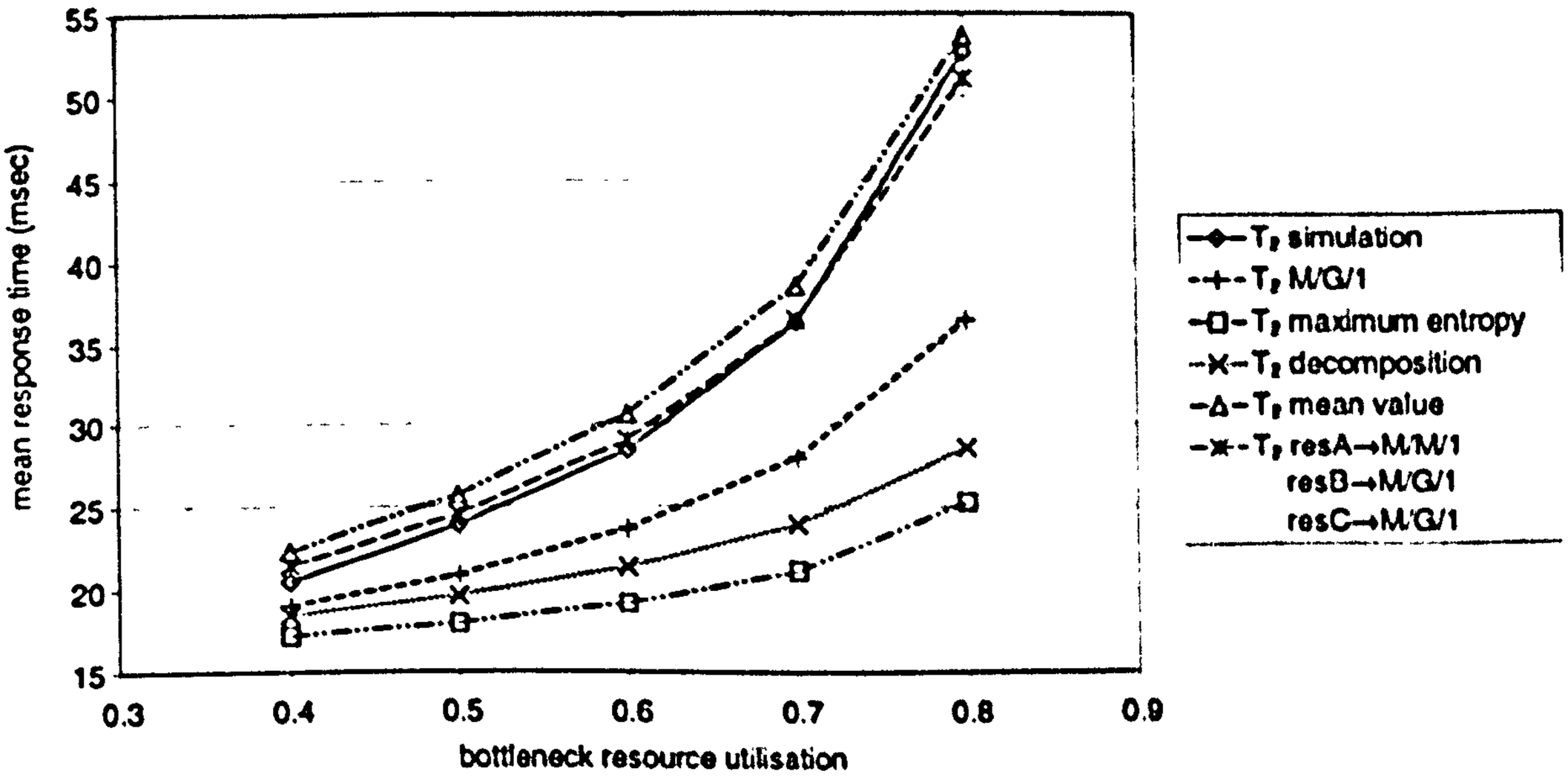


Figure 5-4 Mean response time of T_2 transactions

In Figure 5-5, the mean response time of transaction T from Figure 5-2 is shown. The x and y axis are as before. This time the four discs are the bottlenecks of the network and therefore the disc utilisation is measured on the x-axis. In addition to the other methods, a modification of the maximum entropy method, which initially eliminates the self-loop at the *pu* was used, and the results are included in the figure. This modification is recommended in [104] in order to improve the approximation. In this case the M/G/1 method provides acceptable results that are no worse than those provided by the other methods.

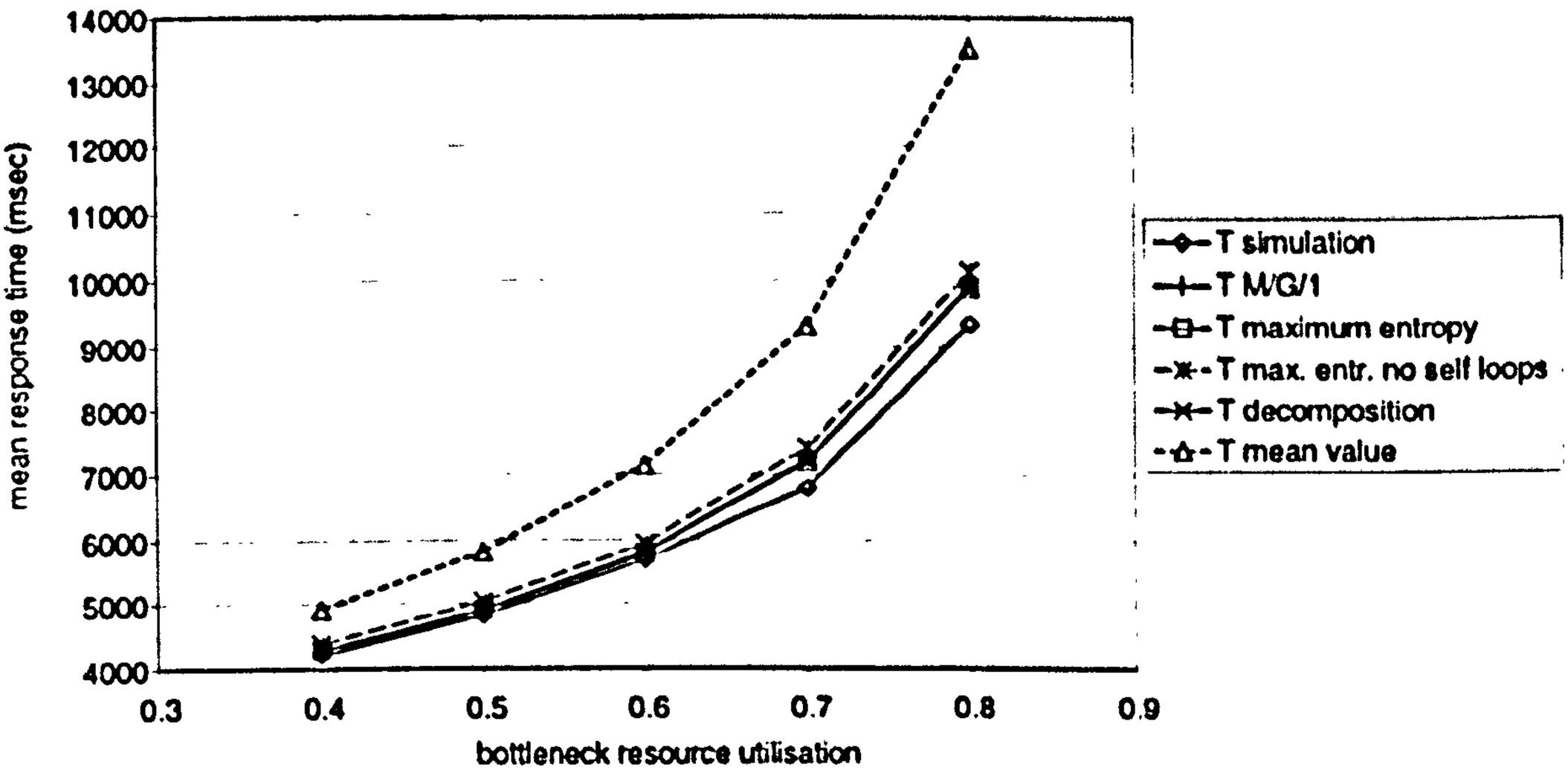


Figure 5-5 Mean response time of transaction of type T

The values obtained using each approach have been compared with those obtained by simulation and the percentage difference for each of the methods for the three transactions are summarised in Table 5-1. This is calculated from the formula $\frac{RT_{appr} - RT_{sim}}{RT_{sim}} \times 100$. The largest value of the percentage difference for each transaction and arrival rate is underlined.

Query	Method	Utilisation of bottleneck resource				
		0.4	0.5	0.6	0.7	0.8
T_1	M/G/1	-1.55	-3.68	-6.27	-10.51	-18.39
T_1	Maximum entropy	-9.46	-14.80	-20.88	-28.64	-39.43
T_1	Decomposition	-3.91	-7.89	-13.04	-20.57	-32.24
T_1	Mean value	14.75	17.23	19.55	20.28	16.71
T_1	M/M/1 / M/G/1	6.56	7.33	8.29	8.40	5.70
T_2	M/G/1	-7.09	-12.51	-17.02	-22.83	-30.67
T_2	Maximum entropy	-16.11	-24.77	-32.72	-41.68	-51.79
T_2	Decomposition	-10.11	-17.65	-25.00	-34.16	-45.50
T_2	Mean value	9.04	7.73	7.70	6.2	2.05
T_2	M/M/1 / M/G/1	4.72	2.69	2.09	0.49	-3.04
T	M/G/1	2.49	2.24	1.94	6.39	6.06
T	Maximum entropy	2.38	2.06	1.69	6.06	5.75
T	Maximum entropy (no loops)	4.52	4.55	4.43	9.02	8.55
T	Decomposition	2.46	2.19	1.89	6.38	6.22
T	Mean value	16.30	20.45	25.20	37.06	44.82

Table 5-1 Percentage difference between results of methods and results of simulation

From these results it would appear that none of the approximation techniques has a consistent advantage over the others in all cases. On the other hand, a good approximation can be achieved by treating the queueing network as consisting of a mix of M/M/1 and M/G/1 resources for these two examples. The reason why a particular combination of M/M/1 and M/G/1 resources leads to a good approximation to transaction response time is explained in Section 5.6; it is also shown how such a combination is to be identified.

5.5 A study of usefulness of techniques

In this section the estimation of response times has been extended to a range of examples to which the approximation methods given in the previous section have been applied. The range includes variations on the two examples already discussed as well as some new transaction types. Examples are numbered using two-digit numbers, indicating class of example (first digit) and id of example within its class (second digit).

The first set of examples consists of variations on the T_1 and T_2 transactions, shown in Figure 5-1. In these examples the service time requirements for the two

transactions are varied while their relative frequencies (0.4 and 0.6) and the number of loop iterations remain constant. In terms of the queueing network in Figure 5-1, the service times and squared coefficients of variation of service time are varied, while the transition probabilities (i.e. the network topology) remain constant. The various examples are given in Table 5-2. Unless shown otherwise, time is expressed in microseconds. Example 01 is the first example discussed in the previous section.

Id.	Service time requirements					Overall squared coefficient of variation of service time		
	T_1			T_2				
	A	B	C	B	A	Cs_A^2	Cs_B^2	Cs_C^2
01	constant 32	constant 60	constant 146	constant 50	constant 100	0.1259	0.0067	0.0
02	exponential mean 32	exponential mean 60	exponential mean 146	exponential mean 50	exponential mean 100	1.2517	1.0136	1.0
03	constant 32	constant 60	hyper-exp. mean 0.146 ms, var. 0.213	constant 50	constant 100	0.1259	0.0067	10.0
04	32, pr. 1/3 20, pr. 2/3	60, pr. 1/5 58, pr. 4/5	146, pr. 1/2 30, pr. 1/2	50, pr. 1/3 12 pr. 2/3	100, pr. 1/8 70 pr. 7/3	0.1452	0.4146	0.4344
05	uniform on [16,48]	uniform on [30,90]	uniform on [73,219]	uniform on [25,75]	uniform on [50,150]	0.2197	0.0907	0.083
06	hyper-exp. mean 0.032 ms var. 0.002	constant 60	constant 146	constant 50	constant 100	0.2	0.0067	0.0
07	constant 32	exponential mean 60	constant 146	exponential mean 50	constant 100	0.1259	1.0136	0.0
08	constant 32	constant 90	constant 146	constant 80	constant 100	0.1259	0.003	0.0
09	constant 32	constant 70	constant 146	constant 60	constant 100	0.1259	0.005	0.0

Table 5-2 Variants based on T_1 and T_2 transactions

Table 5-3 gives a number of different examples based on transaction T from Figure 5-2. In addition to varying the service time requirements for the various resources, the cache miss probability also varies thereby resulting in changes in the topology of the network. Unlike Table 5-2, which shows individual resource usage for each transaction, in Table 5-3 the mean service time for the *pu*, *ssu* and *net* resources are given together with the service time requirements for the four discs. Example 11 corresponds to the example given in Figure 5-2.

Id.	Cache miss probability	Mean service time and squared coefficient of variation			Disc0,1,2,3 service time requirements
		PU	SSU	NET	
11	0.9475	887 1.648	192 0.6061	813 0.0145	s + rd + bll mean 13340
12	0.9475	887 1.648	192 0.6061	813 0.0145	constant 15500
13	0.9475	887 1.648	192 0.6061	813 0.0145	uniform on [10500, 20500]
14	0.9475	887 1.648	192 0.6061	2500 0.019	constant 15500
15	0.9475	887 1.648	192 0.6061	2500 3.15	constant 15500
16	0.9475	887 0.0	1300 0.0	2500 0.019	constant 15500
17	0.9475	887 1.648	192 0.6061	813 0.0145	constant 50500
18	0.5	593 2.705	237 0.36	813 0.0145	uniform on [10500, 20500]
19	0.0525	171 5.1724	319 0.057	813 0.0145	uniform on [10500, 20500]

Table 5-3 Variants based on type T transaction

Two further examples that were tested are illustrated in Figures 5-6 and 5-7. Example 21, shown in Figure 5-6, has three transactions looping over different combinations of the three resources *resA*, *resB* and *resC*.

```
transaction T1                transaction T2                transaction T3
probability 0.3               probability 0.5               probability 0.2

loop {200}{                   loop {100} {                  loop {300} {
  resA 20 µs                    resC 30 µs                    resB 20 µs
  resB 10 µs                    resA 80 µs                    resA 24 µs
  resC 17 µs                    resC 30 µs                  }
  resB 40 µs                  }
}
```

Figure 5-6 An example with three transactions and three resources

The most complex example tested is example 31 given in Figure 5-7. Transaction T_1 is similar to transaction T from Figure 5-2. In this case, however, the cache miss probability is 0.7 and not all four discs have equal probability of being accessed. The second transaction is intended to represent the communication of network packets between two nodes. The *pu* and *ssu* resources of node *A* send 1000 packets via the *net* resource; the packets are received by the *ssu* and *pu* resources of node *B*.

```
transaction T1
probability 0.5

loop [4] {
 pu-A 32 µs;
 ssu-A 60 µs;
 net 146 µs
}
ssu-B 80 µs;
pu-B 56 µs;
loop [186] {
 pu-B 40 µs;
 group {
  pu-B 2551 µs;
  ssu-B 40 µs;
  option {
   disc0 (uniform[10.5, 20.5] ms) : 0.1
   disc1 (uniform[10.5, 20.5] ms) : 0.3
   disc2 (uniform[10.5, 20.5] ms) : 0.5
   disc3 (uniform[10.5, 20.5] ms) : 0.1
  }
 } 0.7
 pu-B 181 µs;
 ssu-B 340 µs;
 net 827 µs;
}
```

```
transaction T2
probability 0.3

loop [1000] {
 pu-A 65 µs;
 ssu-A 80 µs;
 net 146 µs;
 ssu-B 80 µs;
 pu-B 329 µs;
}
pu-B (hyperexp[mean 2.551 ms, var 20])
pu-B 8 µs;
ssu-B 15 µs;
net 37 µs
```

```
transaction T3
probability 0.2

loop [52] {
 pu-B 50 µs;
 ssu-B 15 µs;
 disc0 3241 µs;
 disc0 6482 µs;
}
loop [24] {
 pu-B (exponential[mean 432 µs]);
 ssu-B (exponential[mean 200 µs]);
 net (hyperexp[mean 1.0 ms, var. 4])
}
```

Figure 5-7 Example 31: a complex example

Tables B-1, B-2, B-3 and B-4 in Appendix B give details of the magnitude of the percentage difference obtained when comparing results from the approximation methods to results from simulation runs. For examples 01 - 09 no method is consistently better than the others for both T_1 and T_2 over the complete range of bottleneck utilisation. The only exception is example 02 where all methods give acceptable approximations. The reason is that the service times are exponentially distributed and the network is approaching product form.

Table B-1 reveals that for all examples (apart from 02) for both T_1 and T_2 one of the more easily computed approximations (either the M/G/1 or the mean value approximation) gives better results than the more computationally intensive ones (maximum entropy and decomposition). The latter underestimate the transaction response times, with decomposition performing slightly better. On the other hand, the mean value approximation tends to be an overestimate.

There are only four examples for which a single approximation method provides an estimate that is within 15% of the simulated results for both transactions and over the entire range of bottleneck resource utilisations. This is the mean value method for examples 03, 04 and 07 and the M/G/1 approximation for example 08. There are some cases where the M/G/1 or mean value approximation is acceptable for one of the two transactions, but not for the other. This is the case with the mean value method for T_2 in examples 01, 05 and 06, and the M/G/1 method for T_1 in example 09.

Table B-2 gives the percentage difference for examples 11 - 19. The mean value approximation performs poorly in all cases, consistently overestimating the mean response time of T by up to 57%. On the other hand, the maximum entropy and the decomposition approximations perform much better (except for the high utilisations in examples 17 and 19). The decomposition approximation is marginally better than the maximum entropy one. The simplistic M/G/1 compares very well against both decomposition and maximum entropy.

The percentage differences for T_1, T_2 and T_3 of example 21 are given in Table B-3. The table shows that there is no one method that is best for all three transactions over the 0.4 - 0.8 range. A similar conclusion can be made from the data presented for the complex example 31 in Table B-4. For this example it was anticipated that with the increase in the size of the network and variability of service times of the resources, the quality of the maximum entropy and decomposition approximations would improve. However, no improvement is observed.

5.6 A heuristic rule

In this section a simple heuristic rule is proposed in which each queue in a network is labelled as either an M/M/1 or an M/G/1 resource. The resulting network can then be solved to obtain the response time of the system as a whole.

5.6.1 Definition

The rule is based on the notion of a dominant resource. After experimenting with several definitions for determining dominance, the following has proved simple and successful:

Definition The *dominance* of a resource is a weighted average of its utilisation and its relative visit ratio, i.e.

$$dom(i) = \frac{1}{2} \times \rho_i + \frac{1}{2} \times \frac{e_i}{\sum_i e_i}$$

where the visit ratio of resource i, e_i, is the average number of times resource i is visited for each arrival from outside. Using the notation from Section 3.2.4.1,

$$e_i = \sum_{r=1...R} e_{ir}$$

Since both utilisation and relative visit ratio are numbers in the range 0 to 1, so also is dominance. Although relative visit ratio does not depend on transaction arrival rate, utilisation does. Thus the utilisation of each resource must be taken for some particular transaction arrival rate (e.g. when the bottleneck resource has utilisation of 80%). The heuristic rule is now as follows:

(a) Find the dominance of each resource.

(b) If there is a single dominant resource, i.e. $dom(i) > dom(j)$ for all $j \leq i$, then according to the squared coefficient of variation of service time for each resource, label the resources as follows:

- label the dominant resource i as *max*(M/M/1, M/G/1)
- label all other resources j as *min*(M/M/1, M/G/1)

(c) If there is more than one dominant resource, i.e. $dom(i_1) = \ldots = dom(i_n) > dom(j)$ for all $j \neq i_1, \ldots, i_n$, then label all resources as *min*(M/M/1, M/G/1).

(d) According to the label, use the Khinchin-Pollaczek formula to compute queue waiting time. As discussed in Section 5.1 the formula applies to M/M/1 as a special case of M/G/1.

5.6.2 Intuition behind rule

In the course of the experimentation of the previous section it was observed that in most networks there was a dominant resource, i.e. a resource which 'dictated' the network's performance and ultimately determined its response time. The dominant resource acquired a queue of substantial length when compared to that of other non-dominant resources. When the approximations produced by the various methods were inaccurate, it was because they underestimated the length of the queue for the dominant resource and/or overestimated that for the non-dominant resources.

The rule attempts to remedy this by identifying correctly the dominant and non-dominant resources. The resources are then labelled in such a way that the dominant resource is assigned a high value for its queue length, by choosing between M/G/1 and M/M/1 depending on which one gives the higher value. As discussed in Section 5.1, if the calculated squared coefficient of variation of the service time is greater than 1.0, then M/G/1 gives the higher value; otherwise M/M/1 is higher. On the other hand, each non-dominant resource is assigned shorter queue length by choosing the smaller of M/G/1 and M/M/1. Thus, care is taken to emphasise the importance of the dominant resource, while ensuring that the contribution of non-dominant resources is not over-emphasised. Note that treating a resource with squared coefficient of variation of service time different from 1.0 as an M/M/1 resource, amounts to disregarding the true value of the coefficient. Such an approximation can only work better than an M/G/1 one because the arrival stream to the resource is not Poisson.

The difficulty with the rule has been to find a suitable definition of 'dominant'. The experimentation showed that dominance was not determined simply by the

resource's utilisation. In particular, the amount of network traffic that passed through the resource was an important factor. This is measured by the visit ratio of the resource. A convenient way of bringing together utilisation and amount of traffic was to form a weighted average of the resource's utilisation and relative visit ratio, using a suitable number w, $0 < w < 1$ to weigh the importance of these two factors. With w set to ½ the rule was able to account for all examples considered here. Further experimentation, however, is required to determine whether a different weight would prove more successful, and whether other factors, (for example, the squared coefficient of variation of service time of the resource) ought to be considered when determining the dominance of a resource.

Part (c) of the rule accounts for the case where more than one resource possesses the highest dominance for the network. In this case, the experiments revealed that there is no need to emphasise the importance of any particular resource, and so all resources are given 'shorter' queue lengths. This is achieved by choosing *min*(M/M/1, M/G/1) for both dominant and non-dominant resources. An example where case (c) does apply is the network on Figure 5-2, where the four discs have the same dominance, due to their identical utilisations and visit ratios; it is also the highest dominance in the network. Currently, for (c) to apply, the dominance of such resources is required to be identical. However, further experiments are needed to establish how 'close' $dom(i_1)$ and $dom(i_2)$ must be before they are considered equal and (c) is triggered.

5.6.3 Application of rule

As an illustration of the rule, consider the network given in Figure 5-1 (and all variations on it given in Table 5-2). For the arrival rate for which the bottleneck resource (*resA*) is 80% utilised, the utilisations of the resources are $\rho_{resA} = 0.80$, $\rho_{resB} = 0.50$, and $\rho_{resC} = 0.35$. The visit ratios for resources *resA*, *resB* and *resC* are as follows: $e_{resA} = e_{resB} = 80$ and $e_{resC} = 20$. Since e_{out}, the visit ratio of the outside world, is 1, the

relative visit ratio for resources *resA* and *resB* is 80/(80+80+20+1) = 80/181 = 0.44 while for resource *resC* it is 20/181 = 0.11. Note that, as discussed previously in Section 4.6, the visit ratios of the resources can be computed simply from the resource usage profiles.

In part (a) of the rule the dominance of the resources is computed as:

$$dom(resA) = 1/2 \times 0.80 + 1/2 \times 0.44 = 0.62$$

$$dom(resB) = 1/2 \times 0.50 + 1/2 \times 0.44 = 0.47$$

$$dom(resC) = 1/2 \times 0.35 + 1/2 \times 0.11 = 0.23$$

Since $dom(resA) > dom(resB) > dom(resC)$, part (b) of the rule applies and therefore resource *resA* is labelled as *max*(M/M/1, M/G/1), while resources *resB* and *resC* are labelled as *min*(M/M/1, M/G/1). The squared coefficient of variation is used to determine the larger/smaller of M/M/1 and M/G/1 as follows.

Consider the dominant resource *resA*. It is used 0.4 × 50 = 20 times for 32 μsec each time and 0.6 × 100 = 60 times for 100 μsec each time. Its mean service time (first moment) is therefore

$$\overline{x_{resA}} = \frac{20}{20+60} \times 32 + \frac{60}{20+60} \times 100 = 83,$$

while its second moment is

$$\overline{x^2_{resA}} = \frac{20}{20+60} \times 32^2 + \frac{60}{20+60} \times 100^2 = 7756.$$

From these, the squared coefficient of variation of service time for the resource can be computed as:

$$Cs^2_{resA} = \frac{\overline{x^2_{resA}}}{(\overline{x_{resA}})^2} - 1 = \frac{7756}{83^2} - 1 = 0.126$$

Since the coefficient of variation is less than 1.0, the M/M/1 formula will produce a larger value for the queue length at *resA*, i.e. *max*(M/M/1, M/G/1) = M/M/1. Resource *resA* is therefore labelled as M/M/1. In a similar way, the squared coefficient of variation for *resB* and *resC* can be found and the two resources labelled accordingly.

For example, since there is no variation in the service time of *resC*, its coefficient is 0, and the M/G/1 formula will produce a smaller value for its queue length. Thus *resC* is labelled as M/G/1.

Tables B-5 and B-6 in Appendix B summarise the results from the application of the rule to all the examples described in this chapter. The second column shows the labelling produced by the rule. The rest of the table shows the magnitude of the percentage difference for a particular value of the utilisation of the bottleneck resource. Thus, for example, in the case of example 03, when the bottleneck utilisation is 0.4, 0.5, 0.6 or 0.7, the rule predicts mean transaction response times for T_1 and T_2 to be within 5% of simulated values. When the utilisation increases to 0.8 the value for T_2 is still within 5%, while that for T_1 is within 10%.

From the 165 cases presented, in only 1 (example 17) is the relative error greater than 15%. In this example the service time of the discs is chosen to be much higher than in previous examples. Due to this, relatively long queues build up at the discs. However, by treating them as M/G/1 resources the rule underestimates their queue length. In 156 of the cases (over 94%) the error is within 10%. Moreover, in 112 cases (nearly 68%) it is within 5%.

5.7 Summary

This chapter has studied different approximation techniques, which have been applied to a range of simple examples in which transactions representing simple patterns of resource consumption are modelled with queuing networks, which are not in product form.

In practice, one is not likely to load a database system much beyond 70% utilisation, and 80% utilisation was chosen as a reasonable upper limit beyond which queues rapidly become unmanageable. Thus, for each example a number of different transaction arrival rates were used, corresponding to bottleneck resource utilisations of

0.4, 0.5, 0.6, 0.7 and 0.8. For each of these separate cases the different approximation formulae were used to estimate the queue waiting times.

The study has shown that more sophisticated techniques such as maximum entropy or the decomposition method of Gelenbe and Pujolle perform no better (and frequently worse) than simpler approaches such as the Khinchin-Pollaczek formula or M/M/1 formula. From the results a heuristic rule is developed which outperforms other approximation methods. Its advantages may be summarised as follows:

- The heuristic gives consistently good performance – in over 94% of the cases the predicted values are within 10% of the values obtained by simulation and in nearly 68% of the cases they are within 5%. While other methods may give very good approximation in individual cases, none of them are consistently good over the entire range of examples and arrival rates.
- The heuristic is computationally cheap to apply since it uses quantities such as utilisation, visit ratio, and first and second moments, which are readily available from the resource usage profile of the transaction. By comparison, techniques such as maximum entropy or decomposition require an iterative process, involving re-calculation of several quantities on each iteration and terminating when the values of these quantities have converged.
- The heuristic is explicitly designed to account for a property of the networks, namely, the existence of a dominant resource. Experimentation has confirmed that if dominant and non-dominant resources are identified correctly, a good approximation to the behaviour of the network can be achieved.

In the next chapter, transactions composed from more complex queries, involving pipelines and partitioned execution, are considered. The queue waiting time computed by the heuristic rule is the basis for computing the response time of such transactions.

Chapter 6

ESTIMATION OF TRANSACTION RESPONSE TIME

6.1 Introduction

The third stage of the proposed performance prediction method estimates the mean response time for each transaction in the application. The process of calculating transaction response time is essentially a traversal of the resource block profile. During the traversal, response time is accumulated according to the structure of the blocks and the dependencies among blocks. In particular, account is taken of the templates within each resource usage block (*group*, *option*, *loop*, etc.), the parallelism specified by the block homes (partitioned parallelism), and the dependency (blocking or pipeline) among blocks.

Section 6.2 discusses the response time estimation within a resource usage block, not involved in a pipeline with others. Section 6.3 considers the case of such a block executing in parallel across several nodes (partitioned parallelism). Next, pipelined execution of blocks is considered in Section 6.4. An overall response time for the complete resource usage profile of the query is obtained in Section 6.5. Section 6.6 shows how the response time model is implemented within STEADY. Section 6.7 concludes the chapter with a summary.

6.2 Control structures

Within a resource usage block not involved in a pipeline with other blocks and running on a single processor, the response time of a sequence of two or more resource usage items is the sum of their individual response times. Each of these items may be a simple usage or a template (group, option or loop).

6.2.1 Simple resource usage

The response time of a simple usage involving resource *r* is the sum of the particular service time and the mean waiting time of the resource, as estimated by the rule. Denote this response time as $RT(r)$.

6.2.2 Group template

The general form of a group template with probability *p* is:

group {
    usg_1;
    ...
    usg_n
} *p*

The response time for the group template is computed as:

$$RT(group) = p \times (RT(usg_1) + \ldots + RT(usg_n))$$

6.2.3 Option template

The general form of the option template is:

option {
    usg_1:p_1;
    ...
    usg_n:p_n
}

The response time for the option template is computed as:

$$RT(option) = p_1 \times RT(usg_1) + \ldots + p_n \times RT(usg_n)$$

6.2.4 Loop template (pipelined execution)

The loop template denotes pipeline execution. The response time of a loop usage is computed from the pipeline initialisation time and production time, as described below.

A simple example of a pipeline is given in Figure 6-1 to illustrate the idea. Three resources (*resA*, *resB* and *resC*) are organised in a pipeline that performs 10 iterations. The resources take 3, 4 and 1 unit of time per iteration, respectively. The

times for each resource represent the actual service time added to the mean waiting time for the resource as estimated by the heuristic rule; the service times are not shown. This applies to all subsequent examples.

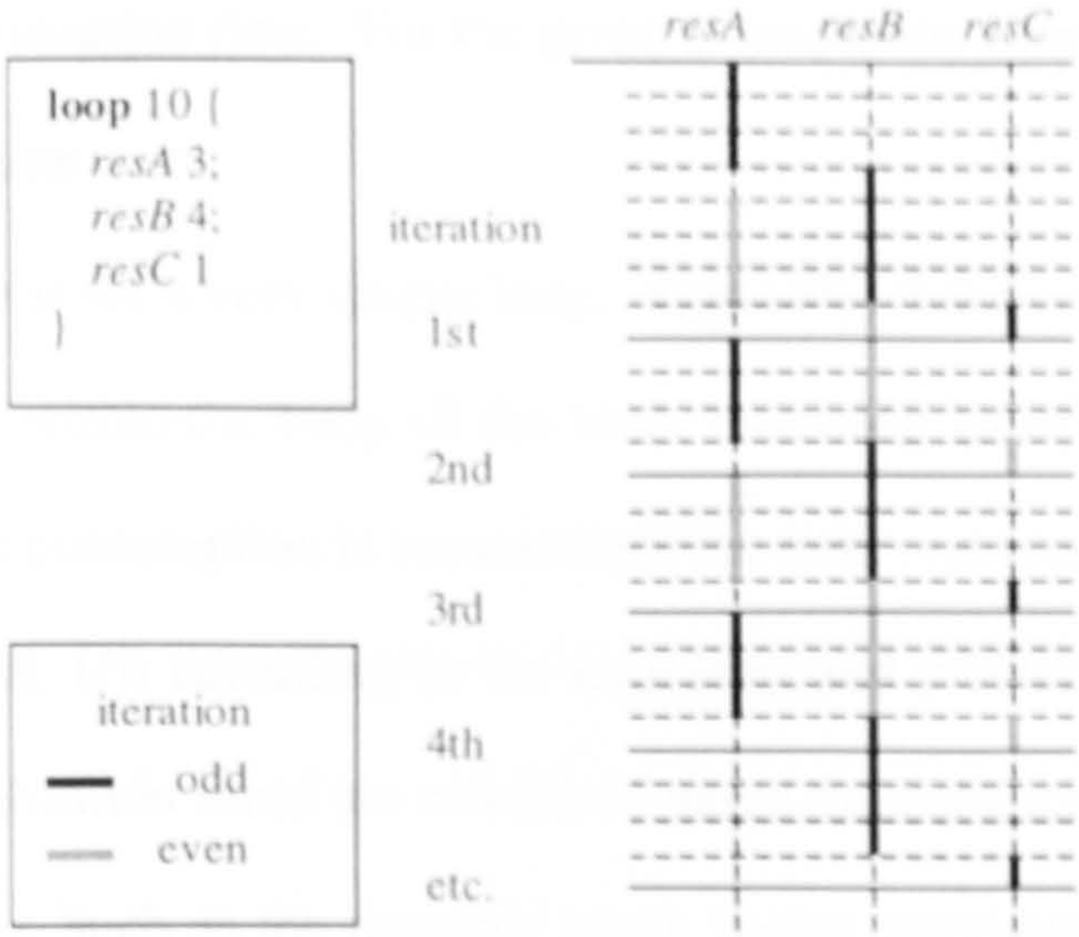


Figure 6-1 Pipelined execution

The execution pattern is presented on the right hand side. The first iteration completes after 3 + 4 + 1 time units. Thereafter, an iteration completes every 4 time units as determined by *resB*, the pipeline bottleneck. This quantity (4 time units) is termed the *production time* of the pipeline and is the highest response time among the resources in the loop.

The time from start to end of an iteration (in this case 3 + 4 + 1 = 8 time units) is termed pipeline *initialisation time*. It is equal to the sum of the response times of all resources involved. The response time of a pipeline loop of n iterations can be approximated as:

$$RT(loop) = \sum_{r \in res(L)} rt(r, L) + (n-1) \times rt(bn(res(L), L), L)$$

where:

- L is the set of resource usage items within the body of the loop;
- $res(U)$ is the set of resources involved in the set of usage items U;
- $rt(r, U)$ is the response time of resource r given its involvement in usage items from U;

- *bn(R, U)* is the bottleneck resource (the one with highest response time) among those in set *R* given their involvement in usage items from *U*.

Informally, the response time is the sum of the initialisation time and the product of *(n – 1)* and the production time. For the given example the formula evaluates to (3 + 4 + 1) + 9 × 4 = 44 time units.

This example is for a very simple loop. Typically, more complicated patterns of resource usage exist within the body of the loop, involving group or option templates. In this case, resource consumption is accumulated for each resource involved in the loop (by *rt(r, U)* and *bn(R, U)*) according to the templates and probabilities involved. The production and initialisation times are calculated based on these accumulated quantities.

In some cases, such as the resource block shown previously in Figure 4-5, one loop template occurs within another. The inner loop is the last template of the outer loop. A simpler example is given in Figure 6-2(a). This case of loop usage is treated as being equivalent to two resource blocks working in a pipeline, as shown in Figure 6-2(b). Section 6.4 discusses the response time of multi-block pipelines in detail.

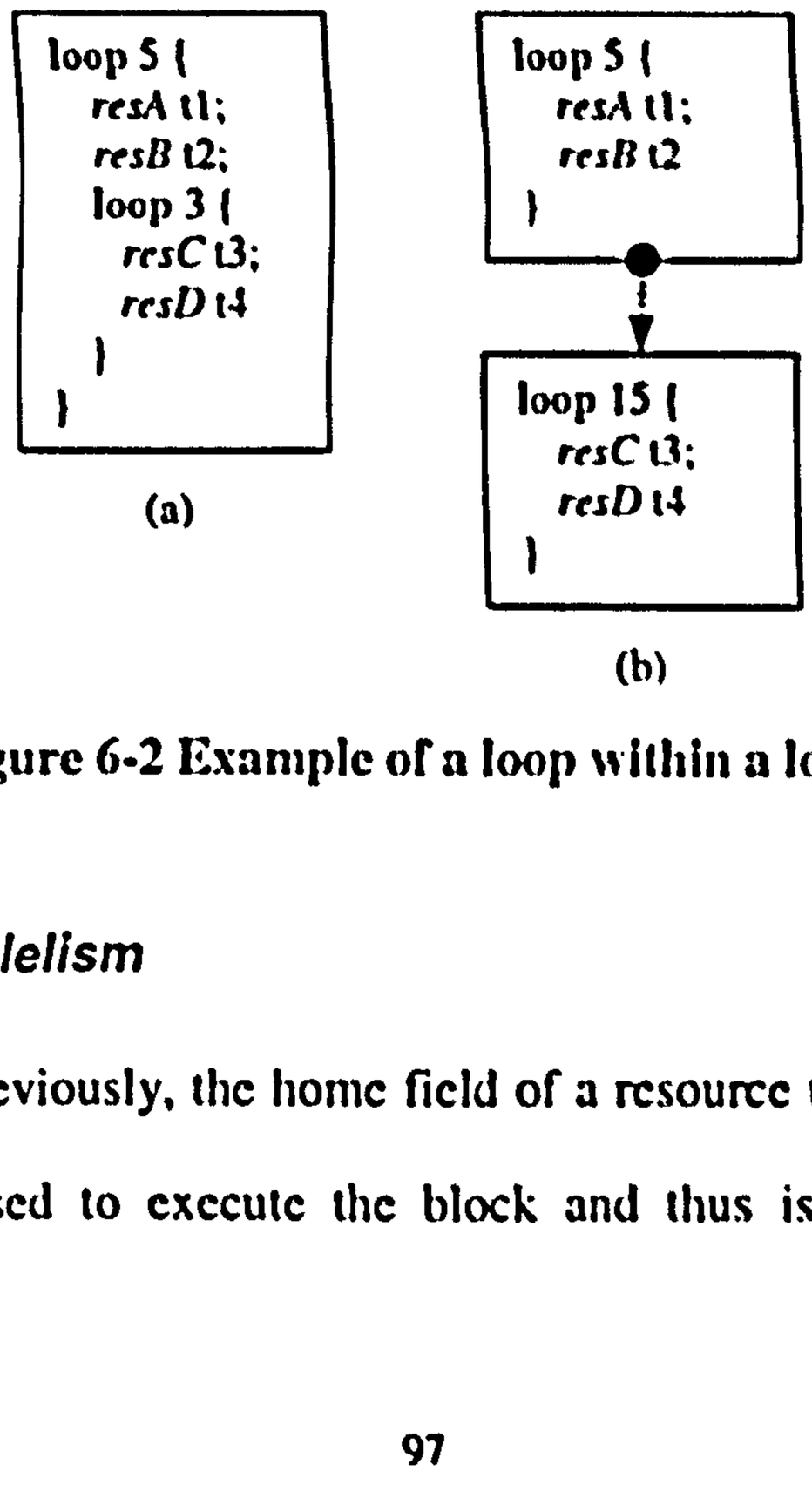


Figure 6-2 Example of a loop within a loop.

6.3 Partitioned parallelism

As discussed previously, the home field of a resource usage block specifies the processing elements used to execute the block and thus is an indication of intra-

operation parallelism. The total response time of a block with intra-operation parallelism is obtained in two steps.

1. First, the response time of each processing element from the block's home is calculated by accumulating time for the sequence of resource usage items in the body of the block. This is done following the procedure outlined in Section 6.2.
2. Second, the processing element with the largest accumulated response time is found. This value is the response time of the block. Since all processing elements in its home are running in parallel, the block will complete when the processing element with the longest response time completes.

When a block with a home comprising more than one processing element is involved in a pipeline with other blocks, the algorithm that analyses multi-block pipelines (next section) deals with the issue of intra-operation parallelism.

6.4 Multi-block pipeline

This pipeline spans two, three or possibly more blocks within a query tree. Typical of the queries for which performance estimation results are reported in the following chapter is a pipeline over two or three blocks.

6.4.1 Simple case

A simple case of a pipeline, which is also useful for illustrating the idea, is that of two single-home blocks in a pipeline with one sending tuples and the other receiving them. An example of two such blocks is shown in Figure 6-3.

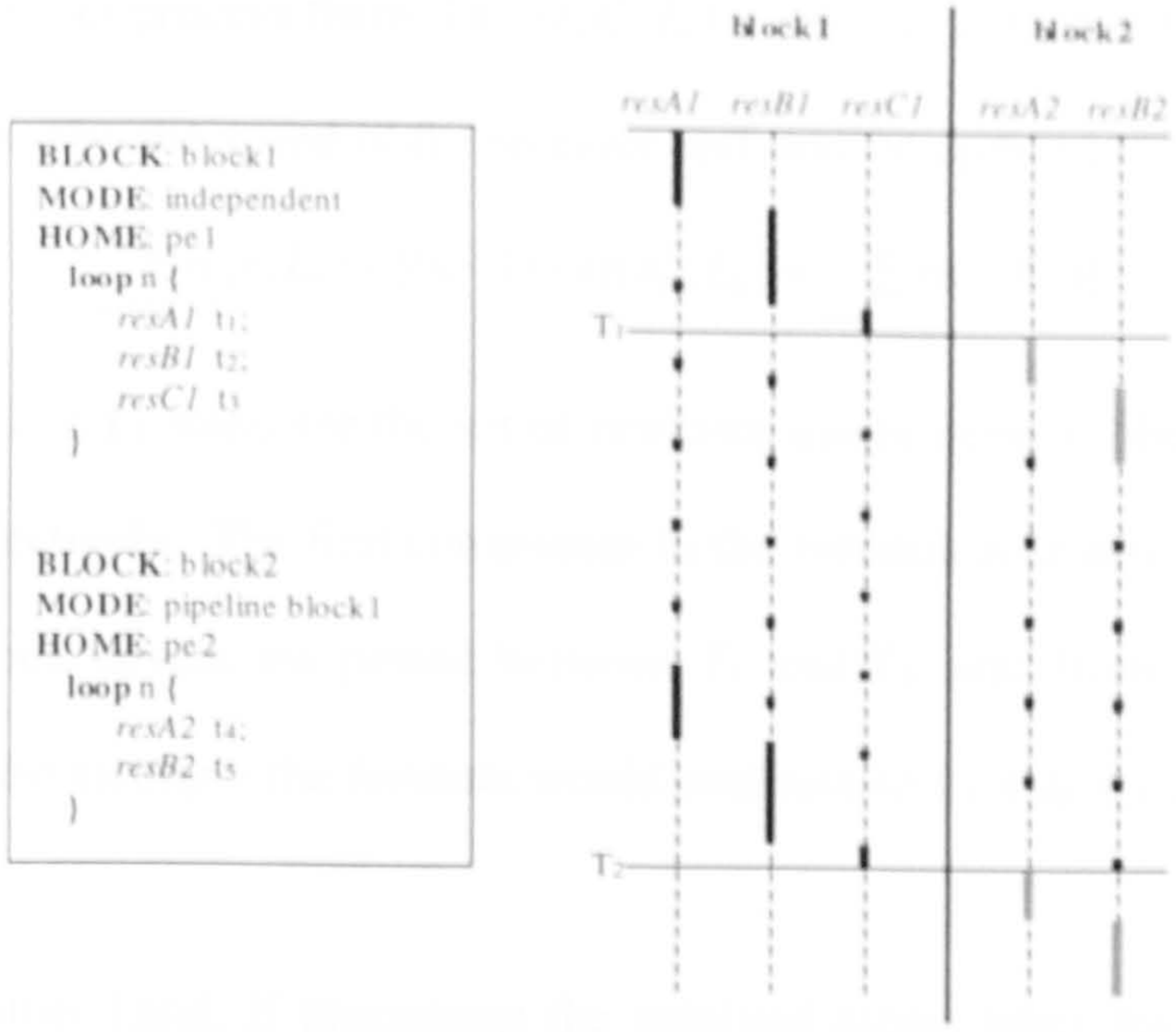


Figure 6-3 Two blocks in a pipeline

PE1 of block1 sends n tuples in a pipeline (in a similar way to the example in Figure 6-1). The sending of a tuple requires work from all three resources *resA1*, *resB1* and *resC1*. Let t_2 be $max(t_1, t_2, t_3)$. The bottleneck resource of this pipeline is thus *resB1*; the pipeline production time is t_2 and the initialisation time is $t_1 + t_2 + t_3$. PE2 in block2 receives the n tuples through the involvement of resources *resA2* and *resB2*. Let the production time of this pipeline be t_5. The full processing time needed to receive a tuple (the initialisation time of this loop) is $t_4 + t_5$.

Time T_1 in Figure 6-3 is the time at which the first iteration of the loop in block1 is complete and the first tuple has been sent to and is about to be received by block2. Similarly, T_2 is the time when the last tuple has just been sent and the work needed for its processing in block2 begins. Between times T_1 and T_2 the sending and receiving process execute together in parallel. The response time of the system depends on the relative speeds of sender and receiver. Let the bottlenecks of the two loops be r_1^* and r_2^*, i.e. $bn(res(L_1), L_1) = r_1^*$ and $bn(res(L_2), L_2) = r_2^*$. Since the resources required by the two blocks are disjoint, if the sending pipeline can send its tuples in less time than

the receiving one can process them, i.e. $rt(r_1^*, L_1) < rt(r_2^*, L_2)$, then the total time of the system is governed by the speed of the receiver and may be approximated as:

$$\sum_{r \in res(L_1)} rt(r, L_1) + [(n-1) \times rt(r_2^*, L_2) + \sum_{r \in res(L_2)} rt(r, L_2)]$$

Here, L_1 and L_2 stand for the set of resource usage items in the loops of block1 and block2, respectively. The first component in the formula accounts for the time prior to T_1. The second covers the period between T_1 and T_2, and from T_2 to the end of execution. For the example the formula would evaluate to $(t_1 + t_2 + t_3) + ((n - 1) \times t_5 + (t_4 + t_5))$.

On the other hand, if processing the received tuples takes less time than their sending, i.e. $rt(r_1^*, L_1) > rt(r_2^*, L_2)$, then the total time of the system is governed by the sender:

$$[\sum_{r \in res(L_1)} rt(r, L_1) + (n-1) \times rt(r_1^*, L_1)] + \sum_{r \in res(L_2)} rt(r, L_2)$$

The analysis is slightly different in the case when the resources required by the two blocks are not disjoint. In this case the calculation of the production time of the each loop must take into account any usage of shared resources taking place in the other loop. Suppose the bottleneck resources of the two loops are r_1^* and r_2^*, i.e.

$$bn(res(L_1), L_1 \cup L_2) = r_1^*$$

$$bn(res(L_2), L_2 \cup L_1) = r_2^*$$

Note that the bottlenecks are determined taking into account resource consumption from both loops. In the case when $r_1^* \neq r_2^*$ a similar analysis to that for the case of disjoint resources applies. In other words, if $rt(r_1^*, L_1 \cup L_2) < rt(r_2^*, L_2 \cup L_1)$, then the response time of the system is determined by that of the receiver and may be approximated as:

$$\sum_{r \in res(L_1)} rt(r, L_1) + (n-1) \times rt(r_2^*, L_2 \cup L_1) + \sum_{r \in res(L_2)} rt(r, L_2)$$

In the case of a slower sender, the overall response time is determined by the sender's production time:

$$\sum_{r \in rm(L_1)} rt(r, L_1) + (n-1) \times rt(r_1^*, L_1 \cup L_2) + \sum_{r \in rm(L_2)} rt(r, L_2)$$

If $r_1^* = r_2^*$ (i.e. the bottleneck is a shared resource for the two loops), the response time of the sender/receiver system is approximated as the sum of the response times of the two loops:

$$\sum_{r \in rm(L_1)} rt(r, L_1) + (n-1) \times rt(r^*, L_1) +$$
$$\sum_{r \in rm(L_2)} rt(r, L_2) + (n-1) \times rt(r^*, L_2)$$

where $r^* = r_1^* = r_2^*$.

6.4.2 Multi-home blocks in pipeline

The idea that the overall time of a sender/receiver system is determined by the slower of the two can also be extended to apply to the case of multiple senders and receivers. This represents the case of a block with a multiple processor home sending tuples to another block executing on multiple processors, with senders and receivers working in parallel. Typically, redistribution of tuples takes place between the two blocks and different number of tuples may be sent/received by different sending/receiving loops, as illustrated in Figure 6-4. The total time of this system is determined by the slowest of all sending and receiving loops.

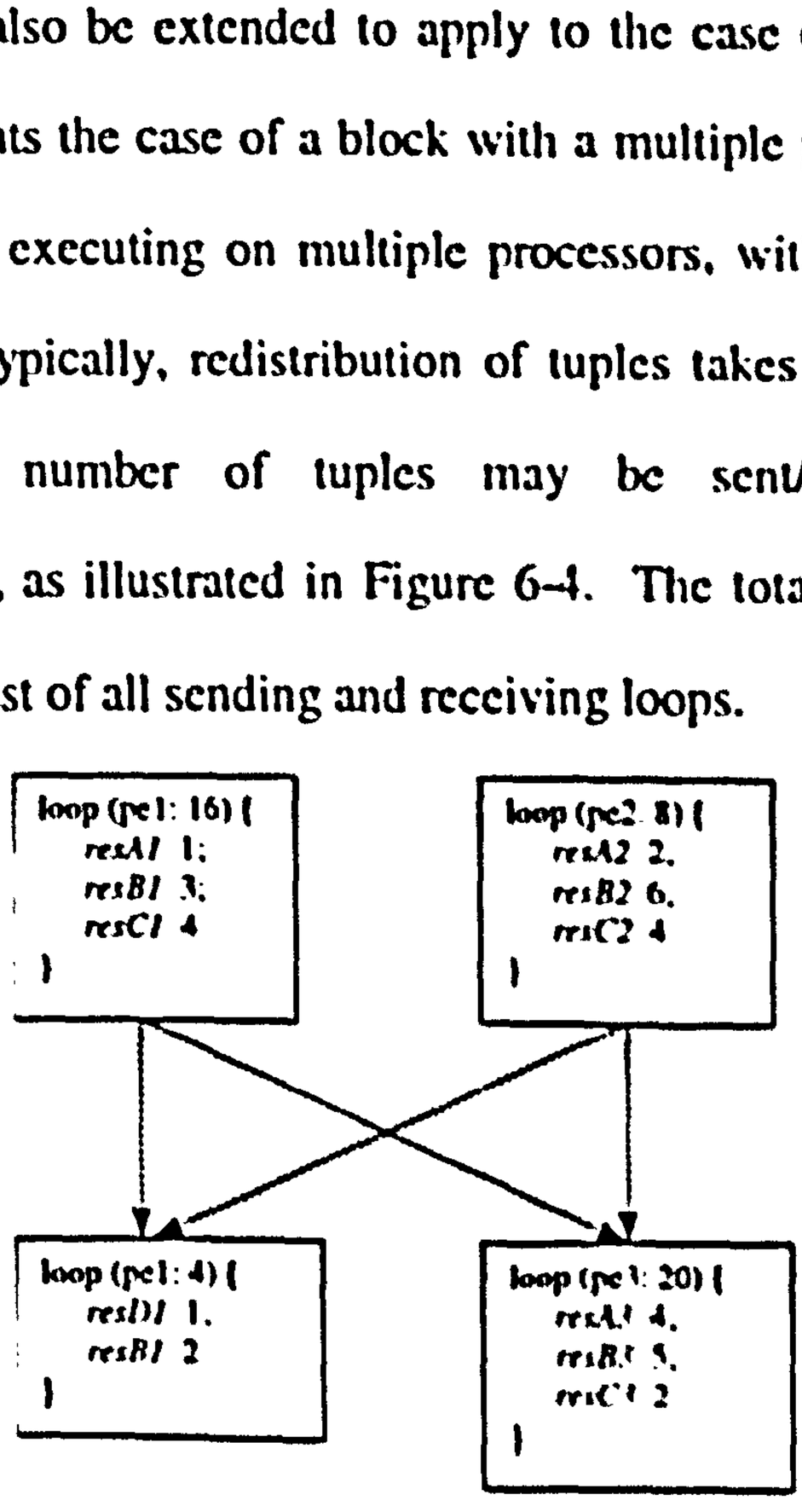


Figure 6-4 Multi-home blocks in a pipeline

First, all sending loops are analysed to determine the slowest one, as follows. Consider sending loop L_S. Suppose there is no receiving loop L_R sharing resources with L_S and let $bn(res(L_S), L_S) = r^*_{L_S}$. The response time of L_S can be approximated as before:

$$RT(L_S) = \sum_{r \in res(L_S)} rt(r, L_S) + (iter(L_S) - 1) \times rt(r^*_{L_S}, L_S) \tag{6.1}$$

where $iter(L)$ is the number of iterations in loop L.

If there is a receiving loop L_R, sharing resources with L_S (there can be at most one such loop, since receiving loops are disjoint), then let $r^*_{L_S}$ be the bottleneck resource of L_S, taking into account any shared use. Formally,

$$r^*_{L_S} = BN(res(L_S), (L_S, iter(L_S)), (L_R, iter(L_R))), \tag{6.2}$$

where $BN(res(L_1), (L_1, n_1), (L_2, n_2))$ returns resource $r_i \in res(L_1)$ such that $n_1 \times rt(r_i, L_1) + n_2 \times rt(r_i, L_2) > n_1 \times rt(r_j, L_1) + n_2 \times rt(r_j, L_2)$ for all $j \neq i$. Then

$$RT(L_S) = \sum_{r \in res(L_S)} rt(r, L_S) + (iter(L_S) - 1) \times rt(r^*_{L_S}, L_S) + iter(L_R) \times rt(r^*_{L_S}, L_R) \tag{6.3}$$

The same analysis can be carried out on each receiving loop, L_R, in order to obtain an estimation of its response time, $RT(L_R)$.

The overall response time of the system of senders and receivers is approximated as follows. Let L_S and L_R be the sending and receiving loops with the highest response times as determined by the above procedure. Let their bottlenecks be $r^*_{L_S}$ and $r^*_{L_R}$, respectively. Suppose that $r^*_{L_S} \neq r^*_{L_R}$. If $RT(L_S) < RT(L_R)$ then:

$$RT = \min_{L_{S_i}} \Big(\sum_{r \in res(L_{S_i})} rt(r, L_{S_i}) \Big) + RT(L_R) \tag{6.4}$$

On the other hand, if $RT(L_S) > RT(L_R)$, then:

$$RT = RT(L_S) + \min_{L_{R_i}} \Big(\sum_{r \in res(L_{R_i})} rt(r, L_{R_i}) \Big) \tag{6.5}$$

In the above two formulae, the minimum initialisation time of sending (resp. receiving) loops is determined on the basis of the assumption that the first (resp. last) tuple to be sent (resp. received) is processed by the "fastest" sender (resp. receiver).

If, however, $r^*_{L_s} = r^*_{L_R}$, then

$$RT = \sum_{r \in res(L_s)} rt(r, L_s) + (iter(L_s) - 1) \times rt(r^*_{L_s}, L_s) + \sum_{r \in res(L_R)} rt(r, L_R) + (iter(L_R) - 1) \times rt(r^*_{L_R}, L_R)$$

Consider the example given in Figure 6-4, which can be used to illustrate the process of estimating the response time according to the above analysis.

Consider first each sending loop. The loop on PE1 shares a resource (*resB1*) with the receiving loop on the same node. The bottleneck resource of this loop ($r^*_{L_s}$), calculated according to (6.2), is the shared resource *resB1*, since its combined usage within both loops is greater than that of the two other non-shared resources (*resA1* and *resC1*). The response time of the sending loop can be calculated from (6.3) as $(1 + 3 + 4) + (16 - 1) \times 3 + 4 \times 2 = 61$.

The sending loop on PE2 does not share any resource usage, and its response time can be calculated from (6.1). The bottleneck resource of this loop is *resB2* (with response time of 6) and therefore the response time of this sending loop is $(2 + 6 + 4) + (8 - 1) \times 6 = 54$

Consider next the receiving loops. Their response times are estimated in a similar way. The receiving loop on PE3 does not share resources with a sending loop and evaluates (via (6.1)) to $(4 + 5 + 2) + (20 - 1) \times 5 = 106$. The receiving loop on PE1, however, shares resource *resB1* with the sending loop on the same node. Resource *resB1* is the bottleneck of this loop and its response time was computed above as 61.

Now consider the sending and receiving loops with the highest response times. These are the sending loop on PE1 with response time of 61 and bottleneck *resB1* and the receiving loop on PE3 with response time of 106 and bottleneck *resB3*. Since the

two bottlenecks are different, and the receiving loop has a higher response time, the overall response time of the system may be approximated according to (6.4) as min(1 + 3 + 4, 2 + 6 + 4) + 106 = 8 + 106 = 114.

6.4.3 More than two stages

This approach can be extended to a pipeline of three or more stages, as follows. First, the slowest stage of the pipeline is identified. This stage could be the initial sending stage of the pipeline, an intermediate sending/receiving stage, or the final receiving stage. The response time of each stage of the pipeline is determined by the response time of the slowest sending/receiving loop within the stage, taking account of shared resource usage as needed. Then, the total time of the pipeline is approximated as the total time of the slowest stage, plus the minimum initialisation times within each of the remaining phases, in a similar way to that explained above.

6.5 Accumulating time

Thus far, the focus has been on estimating response time for various stages of the execution schedule of the query. The response time for the query as a whole can be determined as follows.

The pipeline dependencies among blocks are considered in order to determine which blocks can be analysed together as a single pipeline. The full dependencies are considered to establish when a block (or group of blocks engaged in a pipeline) must wait for some other block to complete before it can begin execution.

Consider the example execution schedule from Figure 4-3. The overall response time of the query is the sum of the response times of three distinct phases within the execution schedule. The first phase is the two-stage pipeline involving blocks SCAN *un80* and BUILD *un80*. The second stage consists of the three-stage pipeline involving blocks SCAN *un60k*, PROBE *un60k*, and BUILD *un80* ⊗ *un60k*. The final phase is the

three-stage pipeline involving blocks scan *un30k*, PROBE *un30k*, and AGGREGATE *un80* ⊗ *un60k* ⊗ *un30k*.

6.6 Implementing the response time model

The response time model has been incorporated within STEADY. The architecture of the enhanced tool is shown in Figure 6-5. The original modules of the tool are shaded in the figure. Based on detailed knowledge of the modelled database system, a new Query Paralleliser was developed within the application layer to transform the query execution plans into parallel execution schedules as described in Section 4.3. The Modeller Kernel was enhanced to transform the execution schedules into task blocks (see Section 4.4). The Evaluator was modified to map the generated task blocks to resource usage profiles as detailed in Section 4.5.

A new response time layer was developed. The Queue Wait Time Estimator estimates queue waiting times for individual resources according to the heuristic rule developed in Chapter 5, while the Response Time Estimator calculates the overall query response time as described in this chapter. The GUI was extended to display the estimated query response times. The Cache Module (Chapter 8) is also shown situated within the DBMS kernel layer.

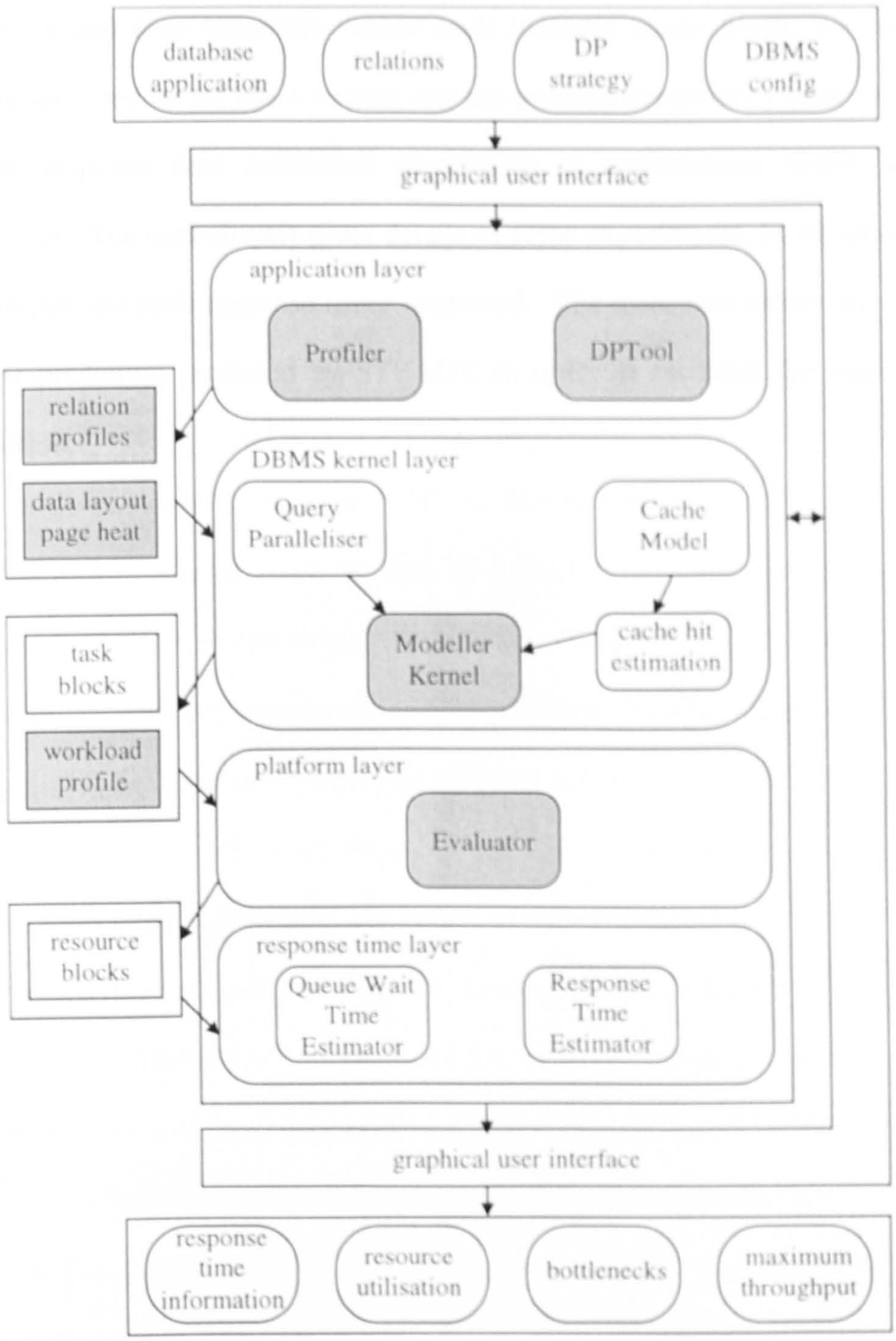


Figure 6-5 Enhanced STEADY architecture; shaded components represent original functionality

6.7 Summary

This chapter has described the way mean transaction response time is approximated through a traversal of the resource block profile of a transaction. As the resource blocks are traversed, response time is accumulated according to the structure of the blocks and the dependencies among blocks. In particular, account is taken of the

group, *option* and *loop* templates within each resource usage block, the partitioned parallelism specified by the block homes, and the pipeline dependency between blocks.

The response time estimation mechanism is implemented within a refined STEADY tool. The next chapter gives details of some experiments, in which a range of queries are run and their response times measured. The measured values are compared against the prediction produced by STEADY in order to establish the quality of the approximation.

Chapter 7

VALIDATION OF APPROACH

7.1 Introduction

The approach to performance prediction presented in the thesis has been validated against measurements from an ICL Goldrush platform running Informix XPS. The aim has been to assess the applicability of the approach as a performance prediction technique and to observe the magnitude of the error in the estimation of measured quantities. In this chapter some results from the validation are presented. The particular Goldrush system used for the experiments reported here is configured with 1 CE, 8 PEs, 6 discs per PE and a cache of 16MBytes on each PE. A number of tables, taken from the AS^{3}AP benchmark (see Section 2.4.2), and queries are used.

The chapter is organised as follows. Section 7.2 provides details of the different tables used in the experiments. In Section 7.3 a range of queries is presented. Included are simple select-project-aggregate queries of the kind found in the AS^{3}AP benchmark, as well as more complex hash-based join queries. The chosen queries are representative of those found in basic decision support applications. The process of calibration of the performance models is discussed in Section 7.4. Through calibration, the costs of basic operations are determined. Section 7.5 describes the process of taking measurements from the Informix XPS/Goldrush configuration. The comparison results are presented in Section 7.6. The chapter concludes with a summary.

7.2 Tables

The *Uniques* relations used are given in Table 7-1. The relations *un270k* and *un540k* are fragmented into 8 fragments by a simple hash function on the *key* primary key attribute with one hash fragment placed on a single disc of each of the eight PEs.

Each tuple has a unique value for attributes *key* and *int*. Attribute *signed*, however, is modified from the benchmark specification so that for each relation tuples have *signed* values in the range 1 to 10 with an equal number of tuples for each value.

Name	Rows	Placement
un80	80	1 disc
un30k	30,000	1 disc
un60k	60,000	1 disc
un90k	90,000	1 disc
un120k	120,000	1 disc
un270k	270,000	1 disc of each PE
un540k	540,000	1 disc of each PE

Table 7-1 *Uniques* relations used for validation

Tables *un30k*, *un60k* and *un90k* are placed on one PE and scans involving only them are not parallelised. Those involving tables *un270k* and *un540k*, however, are performed in parallel by all co-servers containing the table data. The 80 tuples of *un80* are also present in each of the other tables.

7.3 Queries

A number of different queries were considered. They can be classified in the following groups:

7.3.1 Simple select-project-aggregate queries

This group of queries exercises the ability of the method to work for the simple relational operations select, project and aggregate. Different relations (X), predicates (Y) and aggregation operators (max, count, etc) are used, as shown in Figure 7-1.

```
(1)     select max(int)
        from X
        where key > 0;

(2)     select max(int), min(int)
        from X
        where Y;

(3)     select max(int), min(int), avg(int), count(*)
        from X
        where Y;

X = un30k, un90k,
    un270k, un540k;
Y = "signed not in (1)",
    "signed not in (1, 3, 5)"
```

Figure 7-1 Simple select-project-aggregate queries

7.3.2 Simple hash-join queries

Each of these queries finds the maximum value of the *int* attribute from the result of a join of *un80* with the other *Uniques* tables where the *int* values are the same. Each of the other tables contains the tuples of *un80* so the size of the resulting join is 80 tuples.

```
select max(un80.int)
from un80, X
where un80.int = X.int

X = un30k, un90k,
    un270k, un540k
```

Figure 7-2 Simple hash-join query

The query employs a hash-join algorithm, which builds a hash table across all co-servers using *un80*, and subsequently probes this with the tuples from X.

7.3.3 Simple nested query and equivalent non-nested version

This is a simple example of the use of a sub-query. The sub-query is used to return the maximum value of the key attribute from a subset of the tuples of *un30k*. This value is then used in the predicate of the outer query. Note that this is not a correlated sub-query; the execution plan indicates that Informix executes the sub-query first followed by the outer query.

```
select max(int)
from un30k
where int > (select max(key)
             from un30k
             where signed not in (1,3,5));

Y = select max(key)
    from un30k
    where signed not in (1,3,5);
select max(int)
from un30k
where int > Y
```

Figure 7-3 Nested query and equivalent non-nested version

An equivalent "flat" formulation of the query was also considered, as shown in Figure 7-3. The sub-query executes first as an independent query. Following it, a second query uses its result to compute the maximum *int* value. Although equivalent in their semantics, the two queries have quite different response time characteristics.

7.3.4 A union query

This query was used to examine the execution of the Informix union operator when there is opportunity for parallelism. The query groups together the maximum *int* values from tables *un30k* and *un270k*.

```
select max(int)
from un30k
where int > 0

  union all

select max(int)
from un270k
where int > 0
```

Figure 7-4 A union query

It was expected that some form of parallelism would be used when executing the query. However, the execution plans indicate a sequential execution, with *un30k* scanned first, followed by the scanning of *un270k*, and followed by the union operator. To investigate this, the same query was executed with two *un90k* tables, specifically created on different processing elements and discs of Goldrush. Even in this case,

where clearly the query can execute in half the time if the two relations are scanned in parallel, Informix chooses to perform the two scans one after the other.

7.3.5 A three-way hash-join

This is the query whose execution schedule was discussed in Section 4.3 and shown in Figure 4-3. The query is given again in Figure 7-5.

```
select max(un30k.int),
       count(un30k.int)
from un60k, un30k, un80
where un60k.key = un30k.key and
      un30k.key = un80.key
```

Figure 7-5 Joining three relations

7.4 Calibration

Before any comparison against measured performance takes place, the models have to be calibrated. The calibration process seeks to obtain timings for the basic operations that appear in the task block representation of a query. When the task blocks are transformed into resource blocks and each basic operation is mapped to a sequence of one or more resource usage items, the obtained times become the service time requirement of the resources.

For example, a predicate check is a basic operation that maps to a simple PU usage item. This is shown in line 10 of Figure 4-4(a) and line 15 of Figure 4-5. The basic operation send maps to a sequence of three usage items - a PU usage, an SSU usage, and a net usage - as illustrated in lines 12-15 and 18-20 of Figures 4-4(a) and 4-5, respectively.

Some of the costs for basic operations can be obtained by running certain carefully constructed queries in isolation and measuring their execution time with the help of a simple monitoring tool e.g. 'onstat' for Informix XPS. The execution plans of the queries reveal exactly what basic operations are carried out for the query. Thus, the

measured time can be attributed to different phases of the query. The following example illustrates the process.

Suppose the cost (in microseconds) of a non-equality predicate check involving an attribute and an integer constant is to be determined. Firstly, the query

```
SELECT *
FROM um30k
WHERE key < 0
```

is run and its execution time is measured. With all data in cache, the total cost of the query can be expressed using the formula $C(q_1) = 30{,}000 \times scan + 30{,}000 \times pred$. Here, *scan* is cost of accessing a tuple in memory, i.e. moving the tuple from buffer to checking area, while *pred* is the cost of a non-equality predicate check. Next, the query

```
SELECT *
FROM um30k
WHERE key < 0 OR key > 9999999
```

is run. The cost for this query is $C(q_2) = 30{,}000 \times scan + 30{,}000 \times pred + 30{,}000 \times pred$, since two predicate checks are carried out. For both queries no tuples satisfy the predicate, and therefore there is no cost associated with returning tuples. Taking the difference $C(q_2) - C(q_1)$ and dividing by 30,000 gives the cost of *pred*. For Informix on Goldrush this evaluates to 5.74 microseconds. By substituting this value back in the formula $C(q_1)$ the value for *scan* may be obtained. Building on these, more complex queries are used to calculate costs of other basic operations, including aggregation, grouping, updating, deleting and so on. The queries are run independently several times in order to obtain an average for each basic cost. Basic operations such as *send* involve other Goldrush resources: the SSU and the Deltanet. Timing information for these resources, as well as disc service times, were taken from Goldrush specifications provided by ICL. Full details of the calibration process can be found in [113] and [114].

7.5 Taking measurements

A transaction generator was created to emulate a parallel database system workload with many users independently querying the data. It was used to fire queries from the communication element (CE) to a given co-server at a specified rate. Two versions of the generator were written. The first fired transactions with constant inter-arrival times and was used to determine the arrival rate for which the maximum throughput of the machine can be achieved. The throughput was computed by dividing the number of completed transactions by the time between the start time of the first transaction and the end time of the last one. Initially, the measured throughput is equal to the arrival rate at which the queries are fired. As the arrival rate is increased, a value is reached beyond which the measured throughput does not increase further. This is taken to be the maximum system throughput that can be obtained.

The second version of the transaction generator fired transactions with exponentially distributed inter-arrival times and was used to obtain transaction response times. The highest arrival rate used for the exponential generator was equal to the rate at which the maximum throughput (determined using the deterministic generator) was achieved. For each arrival rate the generator was set to fire transactions continuously until 100 transactions were fired. The response time of each query was recorded, and the average was formed to compare against the value estimated by the analytical method. At the higher arrival rates, when queues would build quickly and the system would take longer to settle into a steady state, more than 100 queries were run when needed.

Care was taken to ensure that throughout the process of taking measurements no user processes other than the DBMS software itself were running on Goldrush and that no connections to co-servers other than the one from the transaction generator was maintained. Thus the obtained measurements capture only the activity due to the execution of the specific queries fired at the system. Moreover, since the queries used

do not modify data, background activity, such as flushing of dirty buffers or performing a checkpoint, should not take place and influence the measurements.

7.6 Results

This section presents some results from a comparison of estimated versus actual performance. Note that the performance is estimated under the assumption that there are no delays due to lock contention.

7.6.1 Simple select-project-aggregate queries

Figure 7-6 shows the maximum throughput prediction against actual measured throughput for one of the queries. The chosen query is of type (1), as shown in Figure 7-1. The x-axis represents arrival rate and the y-axis – throughput. Note that STEADY's prediction is a single figure, independent of the arrival rate. The predicted maximum throughput is higher than the one achieved by the system. This can be attributed to additional operating system overhead for which no account has been taken in the model, or to the inaccuracies in the measured basic costs. Despite this, the predicted maximum throughput is an acceptable upper limit of the achieved maximum system throughput.

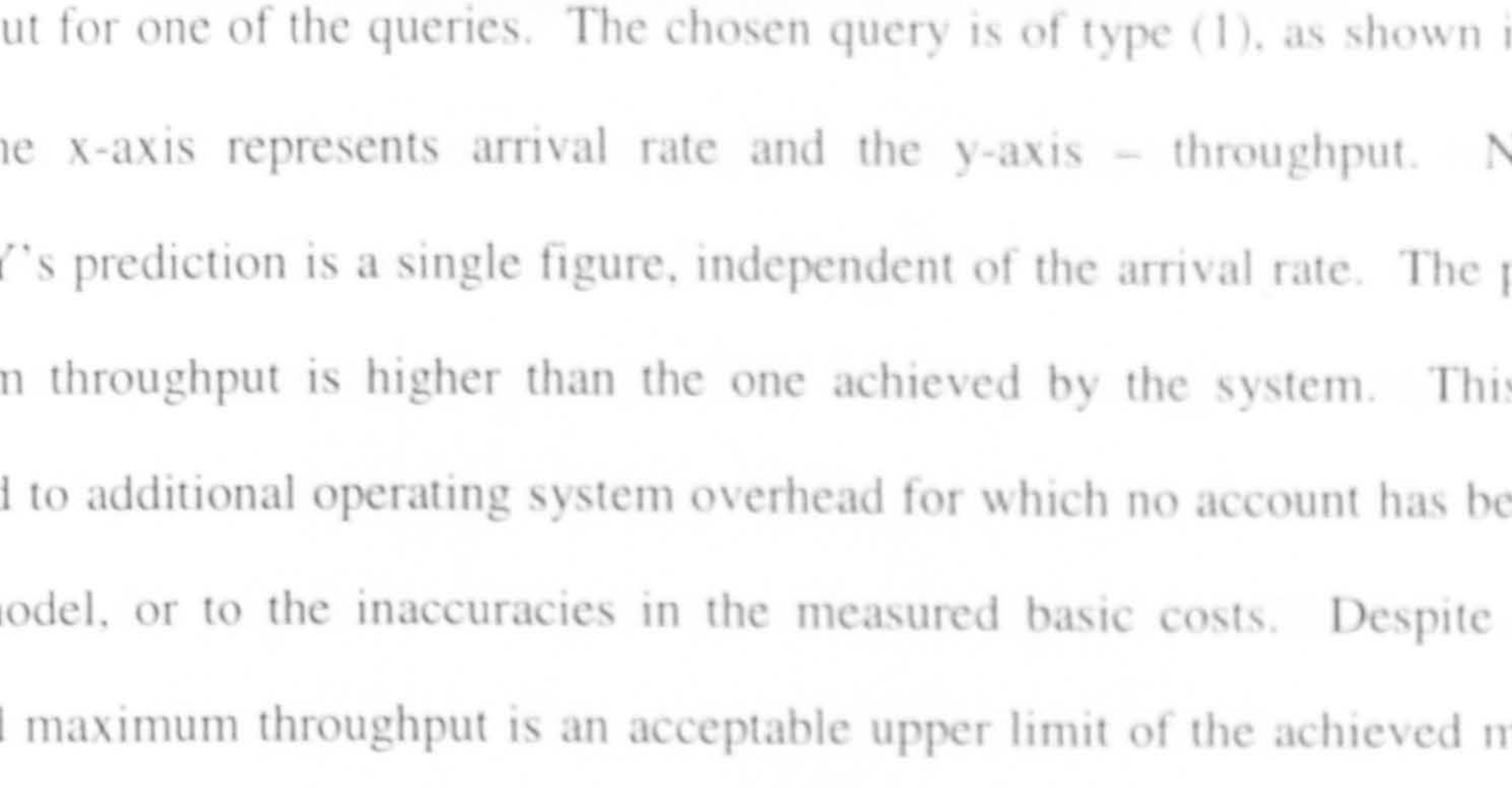


Figure 7-6 Throughput for type (1) query on *un90k*

The response time predictions for this query are shown in Figure 7-7. The x-axis measures arrival rate, while the y-axis represents the measured and predicted

response time. The prediction compares well against the measured value for the entire range of arrival rates.

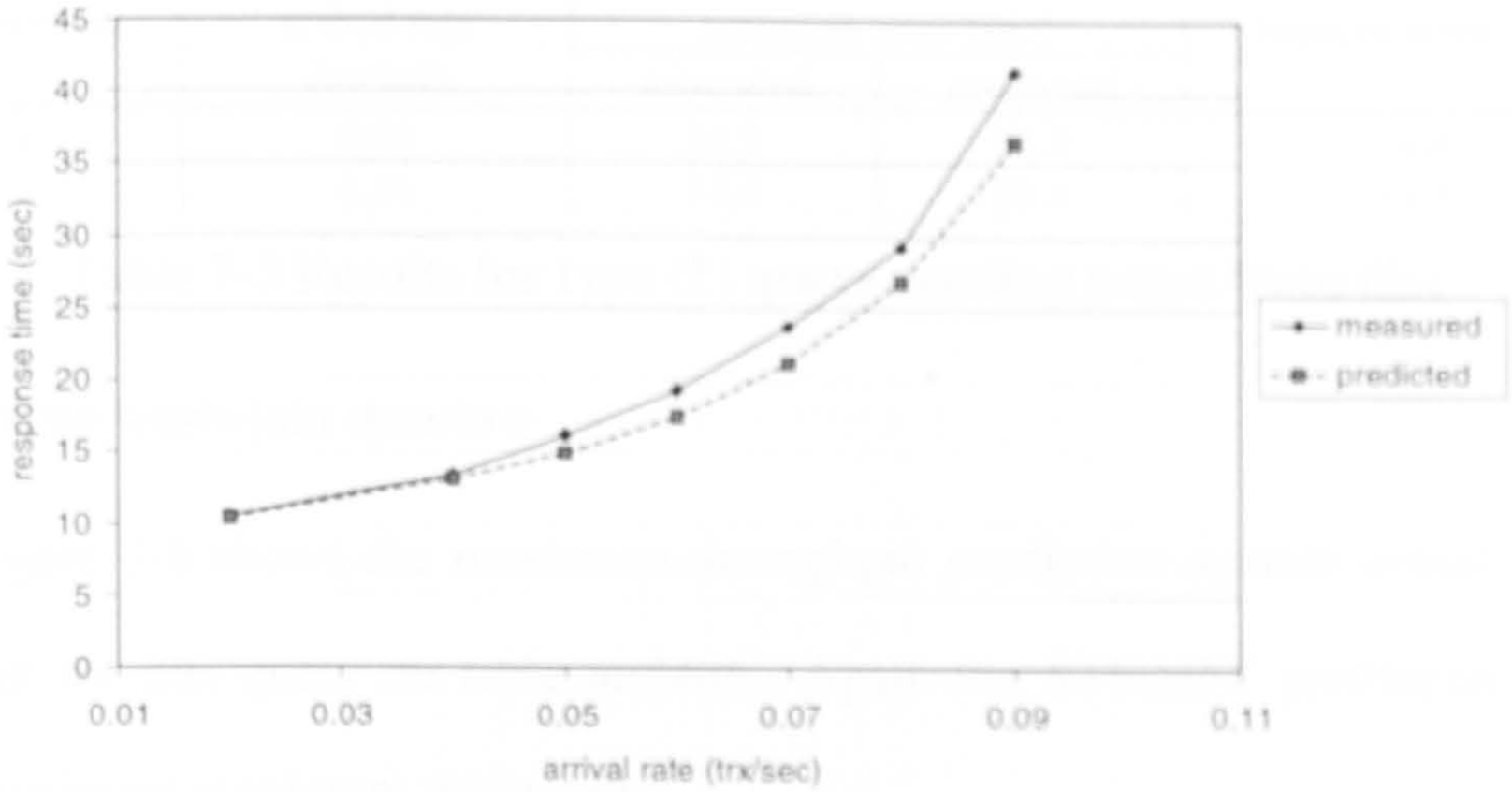


Figure 7-7 Response time for type (1) query on *un90k*

The results of the response time prediction method for all queries of type (1), (2) and (3) from Figure 7-1 are summarised in Table 7-2. The table gives the number of experiments and maximum and average of the relative percentage error computed from the predicted and measured response time. The experiments are divided into two groups depending on the magnitude of the arrival rate for the experiment. The first category is for arrival rates that are within 60% of the predicted maximum throughput, while the second is for rates between 60% and 80%. For example, 11 different experiments with a query of type (1) were performed, each using a different arrival rate not exceeding 60% of the maximum throughput. The largest relative error for this set of 11 experiments was 17.5%; the average was 9.8%.

query	relative error (%)					
	0% - 60% max throughput			60% - 80% max throughput		
	number of experiments	max	avg.	Number of experiments	max	avg.
type (1)	11	17.5	9.8	5	22.7	13.6
type (2)	21	19.4	9.1	9	28.9	15.5
type (3)	15	18.9	12.0	8	19.8	12.2

Table 7-2 Summary of results for simple select-project-aggregate queries

All the experiments reported here were performed with the data present in cache. Some results are also presented for a query of type (1) using table *un120k*, for which all data pages are read from disc. The results from the only two experiments performed are

presented in Table 7-3. The arrival rates for the two experiments correspond to 40% and 80% of maximum throughput.

experiment	arrival rate (trx/sec)	response time (sec)		relative error (%)
		measured	predicted	
1	0.03	24.9	23.9	-4.0
2	0.06	74.8	83.4	11.5

Table 7-3 Results for type (1) query reading pages from disc

7.6.2 Simple hash-join queries

Figure 7-8 shows the maximum throughput prediction against actual measured throughput for this query on table *un540k*. Again the STEADY prediction is higher than the achieved maximum throughput.

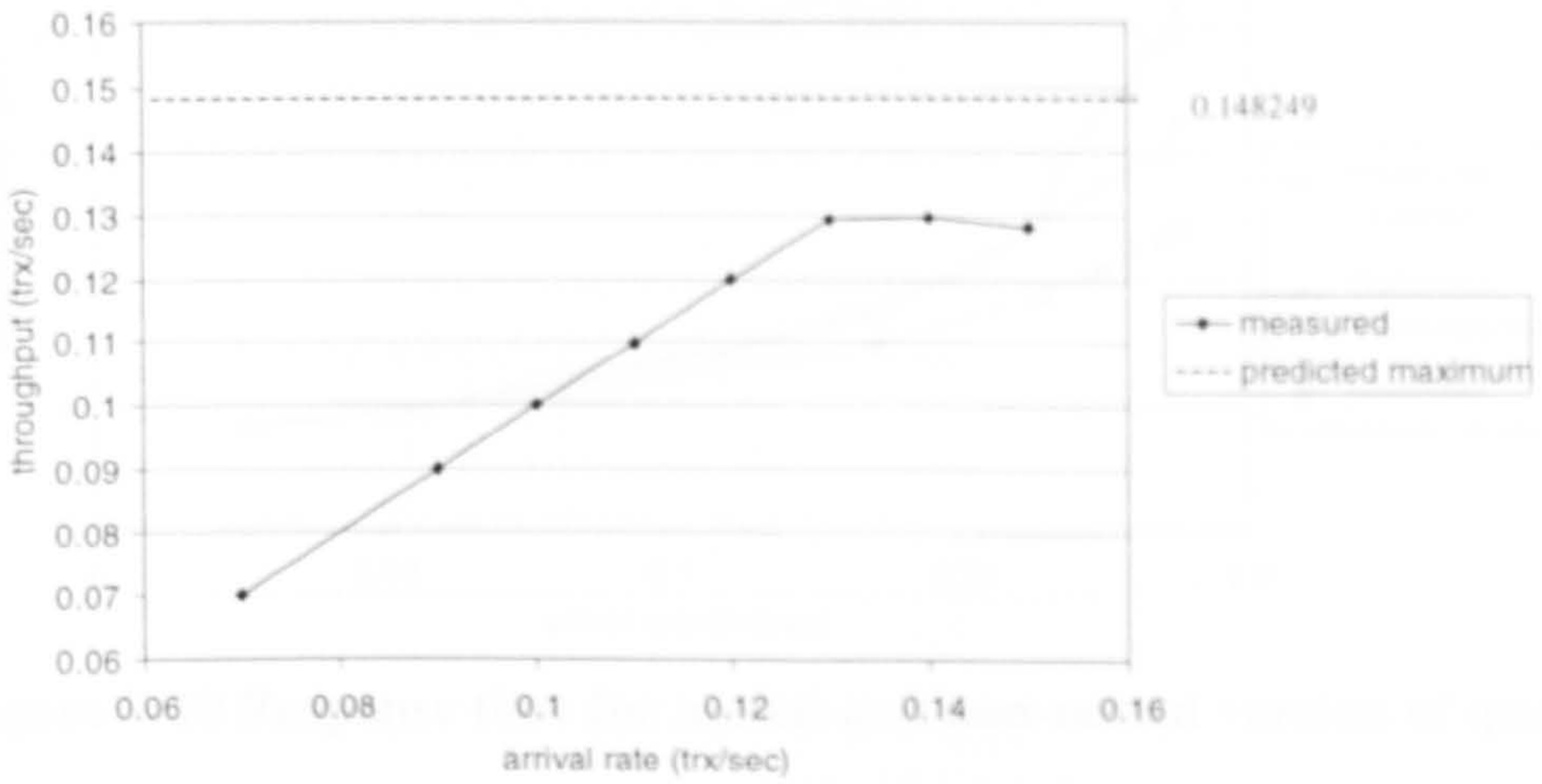


Figure 7-8 Throughput for simple hash-join query on *un540k*

Figure 7-9 shows the response time prediction compared against measured response time. Again, the predicted and measured values appear close.

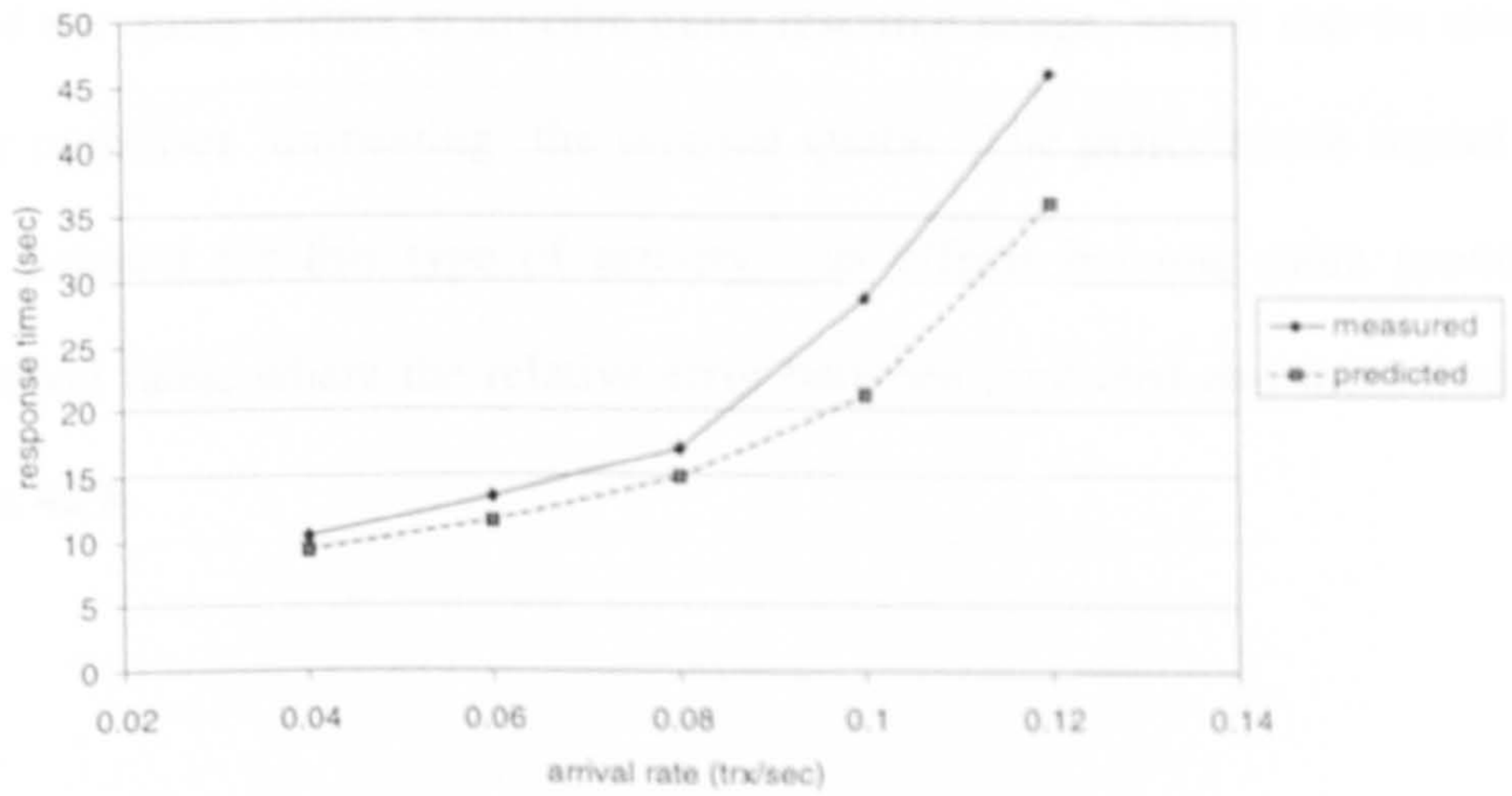


Figure 7-9 Response time for simple hash-join query on *un540k*

The results from all simple hash-join experiments are summarised in Table 7-4.

query	relative error (%)					
	0% - 60% max throughput			60% - 80% max throughput		
	number of experiments	max	avg.	Number of experiments	max	avg.
simple hash-join	8	18.3	9.4	6	28.1	17.0

Table 7-4 Summary of results for simple hash-join queries

7.6.3 Simple nested query and equivalent non-nested version

The graph in Figure 7-10 shows the response time prediction and the corresponding measured values for the nested and non-nested version of this query.

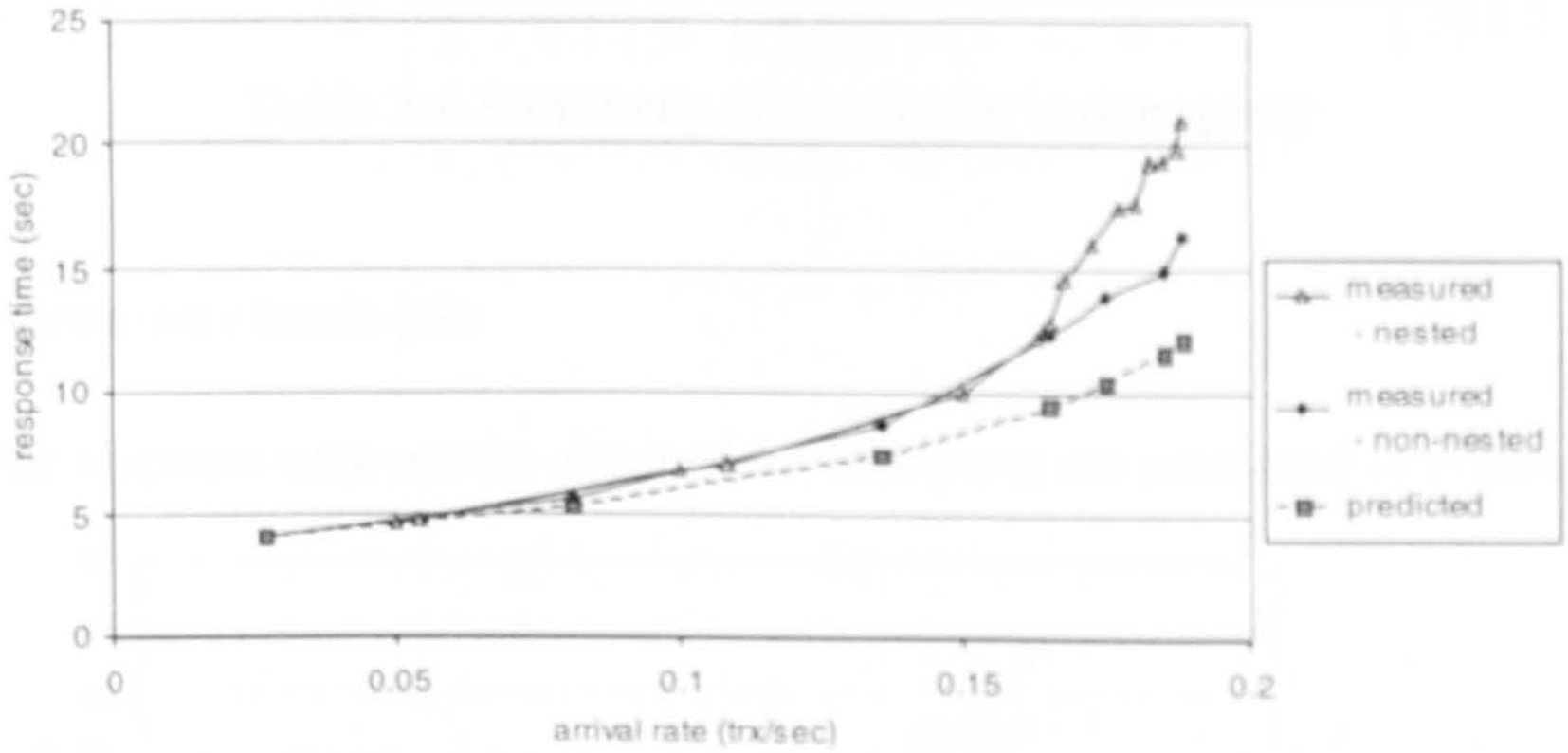


Figure 7-10 Response time for nested and non-nested version of query

Table 7-5 presents the relative error computed from the estimated and measured response time values for both the nested and non-nested values. Both the figure and the table indicate that STEADY compares better with the non-nested query. The nested version of the query seems to involve extra resource usage, which can be attributed to the query optimiser 'un-nesting' the original query. The performance model does not currently account for this type of activity. Its effects become more pronounced at higher arrival rates, where the relative error between predicted and measured values is as high as 42%.

Query	relative error (%)					
	0% - 60% max throughput			60% - 80% max throughput		
	Number of experiments	max	avg.	number of experiments	max	avg.
nested	5	17.2	9.5	10	42.3	36.1
non-nested	5	15.3	5.3	11	24.5	20.1

Table 7-5 Results for nested and non-nested query

7.6.4 A union query

The results for the union query are given in Table 7-6.

query	relative error (%)					
	0% - 60% max throughput			60% - 80% max throughput		
	number of experiments	max	avg.	number of experiments	max	avg.
union	3	10	6.5	5	29.8	27.8

Table 7-6 Summary of results for union query

7.6.5 A three-way hash-join

The response time results obtained for this query are presented in Figure 7-11.

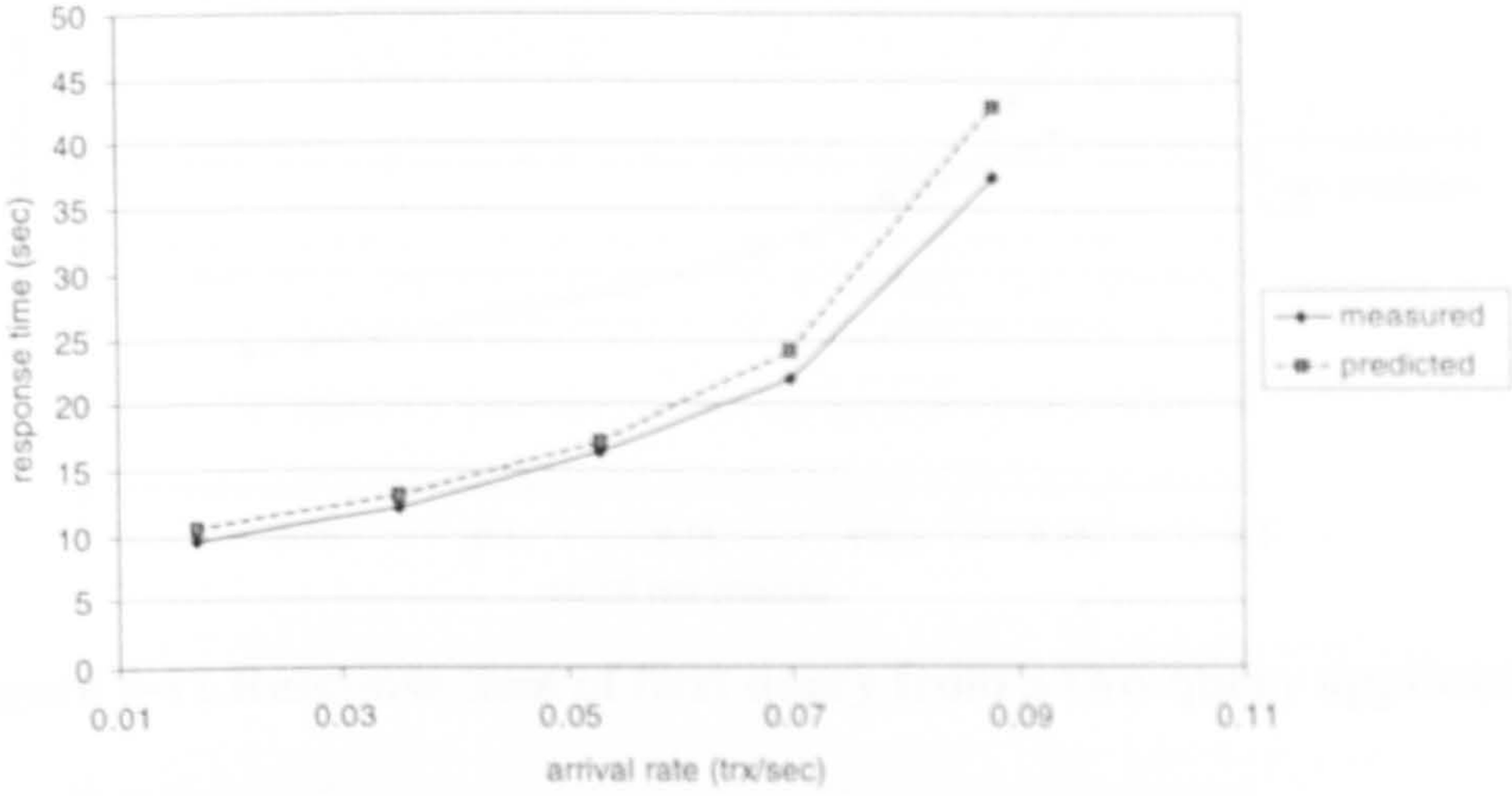


Figure 7-11 Response time for a three-table hash join query

Again, the relative errors are summarised in Table 7-7.

query	relative error (%)					
	0% - 60% max throughput			60% - 80% max throughput		
	number of experiments	max	avg.	number of experiments	max	avg.
triple join	3	9.7	7.7	2	14.5	12.3

Table 7-7 Summary of results for a three-table hash join query

7.6.6 A two-query application

The results of a two-query application are also presented. The two chosen queries are a type (3) query on relation *un30k1* with predicate 'signed not in (1, 3, 5)' and a type (1) query on relation *un30k* (see Figure 7.1). The queries are assigned different frequencies: 70% for the first and 30% for the second. The measured and approximated response times of the two queries are presented in Figure 7-12 and Figure 7-13. There is a reasonable match between the approximation and the measured values.

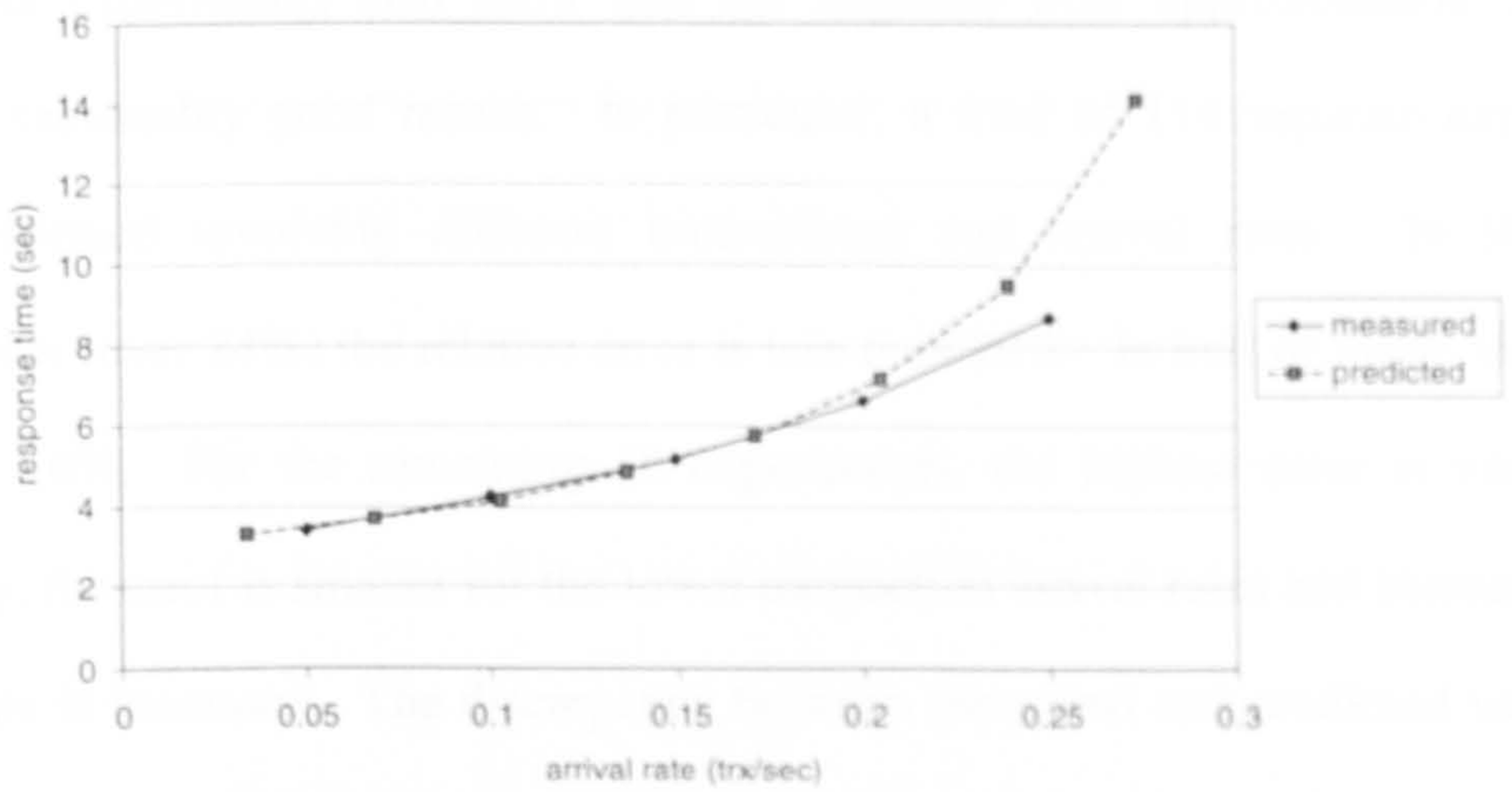


Figure 7-12 Response time of first query from a two-query application

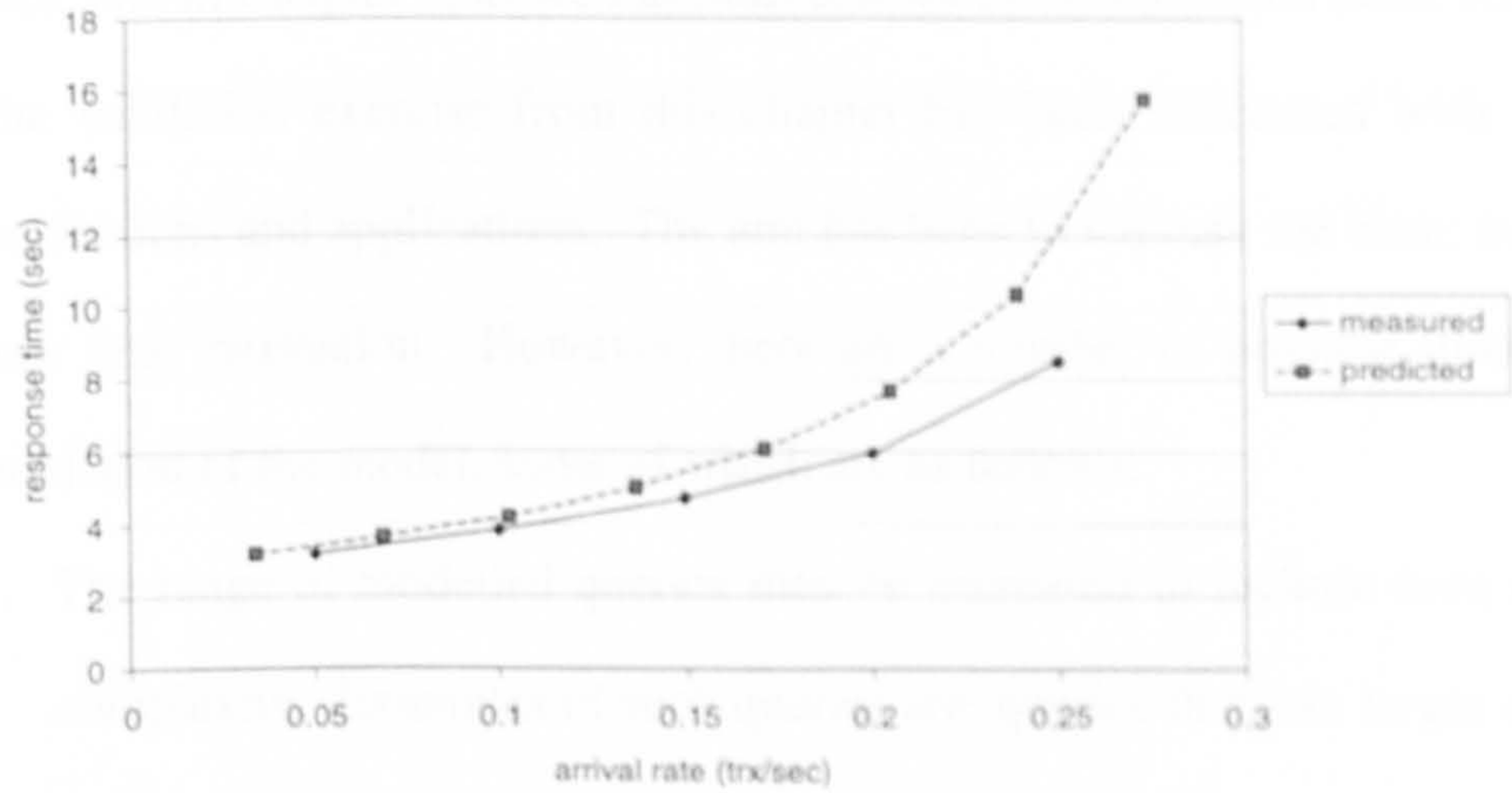


Figure 7-13 Response time of second query from a two-query application

7.7 Summary

This chapter has presented the results from a range of experiments designed to validate the performance estimation techniques developed in Chapters 4, 5 and 6. The experiments cover a number of different tables, queries and relational operations.

The experiments indicate that the estimated and measured quantities are in reasonably good agreement. In particular, the predicted maximum throughput is typically higher than the measured maximum throughput by up to 15%. A possible explanation for this is additional resource consumption due to activity by non-DBMS processes or the operating system itself. Such activity is not accounted for within the models at present. Despite this, the approximation to the maximum system throughput is acceptable.

The experiments also show that the response time approximation technique produces reasonably good results. In particular, a total of 114 separate experiments were performed involving different transactions and arrival rates. In 96 of the experiments (over 84%) the relative error is less than 20%. In half of these, the error is less than 10%. For the remaining 18 experiments, the highest error is under 30%. Generally, the error is smaller for the lower transaction arrival rates and increases as the arrival rate is increased. The discrepancy between measured and predicted values may be partly attributed the approximations introduced within the analytical model itself, e.g. the heuristic rule in Chapter 5, as well as inaccuracies in the measured basic costs.

The validation exercise from this chapter has been concerned with relatively simple transactions and applications. The aim has been to validate the basic mechanism of response time estimation. However, there are a number of possible directions for further validation of the model, some of which are as follows:

1. The range of modelled queries may be increased to include ones of greater complexity. Examples of such queries are: queries that join larger number of tables and/or use different join algorithms; queries that sort data (e.g. use the

ORDER BY and GROUP BY clauses); and queries which use advanced techniques such as correlated sub-queries. First, the model must be calibrated with appropriate basic costs for such operations. Queries of different complexity may then be grouped into multi-query transactions. These, in turn, may be grouped into applications to reflect a more realistic environment. By specifying distinct frequencies for different transactions the complexity of the resulting workload may be controlled and the ability of the model to predict accurately the response time of transactions from complex applications may be verified.

2. Transactions that modify (i.e. update and delete) data may be included in the modelled workload. The execution of such transactions will trigger other DBMS activity, such as flushing of buffers, executing checkpoints and logging. In order to validate the model in such an environment, performance models of background database activity, as well as models of concurrency control and cache maintenance, need to be developed. It may not be possible to obtain measurements for each such model in isolation from the others. Therefore, the validation in an update-intensive environment is a challenging task.
3. Independently of the queries and applications, the machine and database configuration may be varied in order to experiment with the model in different conditions. For example, the size of the database tables, as well as the nodes and discs on which they are placed may be changed. The effects of changing the size of the database cache at each node may also be investigated.

Chapter 8

MODELLING CACHE

8.1 Introduction

In the process of normal query execution, parallel database systems perform other supporting (or background) activity. This includes flushing of database pages, recovering from failure should it occur, ensuring the consistency of data by maintaining locks on data items, and so on. In a complete performance model all such forms of background activity should be accounted for, since they influence the performance of the system. Modelling all types of background activity is beyond the scope of the thesis. However, one particular type of activity – cache maintenance – has been considered in some detail.

The maintenance of a cache for recently accessed database pages can greatly reduce query execution time. With a database cache, not every logical I/O operation implies a physical one. Therefore, capturing the effects of the cache is essential for accurate performance modelling. This chapter introduces cache models for Oracle and Informix, both of which have been implemented within STEADY. The models aim to predict the probability with which pages of different database tables are to be found in cache, given various information, such as the sizes of tables and cache, the cache coherency policy used and the set of transactions accessing the data. Thus estimated, the cache-hit probabilities can be associated with the read operations within the task block (and, therefore, resource usage block) representation for the given transactions, and taken up in further performance analysis.

Section 8.2 provides background and previous work on cache management. Section 8.3 gives details of the Oracle parallel cache management strategy and proposes

an analytical model of it. This was done to illustrate how a model of the cache management mechanism of the system can be incorporated into the overall representation approach. Similarly, a model for Informix is proposed in Section 8.4. Some implementation issues are discussed in Section 8.5. Finally, some results from the Oracle cache model are presented in Section 8.6. A comparison with results from Informix is also shown.

8.2 Background

As mentioned previously, in shared disc systems, each of the nodes of the parallel machine may access and modify data residing on the discs of any of the nodes of the configuration. In shared nothing systems, each node owns the portion of the database that resides on its discs, and it cannot directly read or modify data owned by other nodes. In both cases the nodes maintain caches which store copies of database pages, recently accessed by queries running on the node. This reduces I/O activity at the node and, in shared disc systems, remote data access. However, for parallel database systems utilising the shared data model it introduces the *cache coherency problem* [115]: since the same database page(s) may be cached at more than one node at any one time, cache coherency policies must be used to ensure that queries only see current data. There are a number of cache coherency policies developed in the literature, and a number of studies comparing their performance.

Cache coherency policies are studied in two somewhat different multi-system database environments. In the case of a *client/server* database architecture, the database resides on the server, while applications programs run on client workstations and access the database by communicating with the server. Caching a portion of the database on the client may reduce the number of messages between client and server. A cache coherency protocol is used to ensure that each client's cache remains consistent with the shared database. Wilkinson and Neimat [116], and Wang and Rowe [117] address this

problem, and detail several cache coherency policies. Common to them is the fact that they are integrated with the concurrency control policy of the DBMS.

In the case of a *parallel* database architecture using the shared data model, the database is accessible from each node in the system. The nodes maintain caches that store recently accessed database pages. A cache coherency policy must ensure that database pages in several caches are up to date. This means that a page updated by a transaction on one node must either be propagated to all other nodes, or all nodes that have copies of the page in their caches must invalidate them.

In [118] six coherency policies are examined and compared. They fall into three categories. Under the *notification* of invalidated pages category, the updating node sends the identity of the updated pages to remote nodes at transaction commit time, so that they may invalidate old copies of the pages, if they are present in their caches. Under the *detection* of invalidated pages category no messages are sent, but the validity of a page is established by the accessing node at page access time. Under the *propagation* of invalidated pages category, the updated pages themselves are propagated to the remote nodes at transaction commit time. Common to these policies is the assumption that the DBMS operates under the *force* scheme: at the transaction commit point all updates are propagated to disc and all associated locks are released.

Several cache coherency policies have been proposed that support a *no-force* or deferred write scheme. Under this scheme the propagation of updated database pages to disc is delayed beyond the transaction commit point and locks on the pages are retained. While increasing the recovery complexity, this has the advantage of reducing the number of disc writes. However, care must be taken to guarantee cache coherency in cases when an updated page which has not yet been written to disc (referred to as a dirty page) is needed by another node. Mohan and Narang [115] propose several coherency policies based on the no-force scheme and exclusive lock retention. Similarly, in [68] Dan and Yu give details of five coherency policies and compare their performance

through analytical models. Two of the policies are based on the force scheme, while the remaining three retain locks on cached database pages. The policies differ in the degree of database recovery complexity that they impose. However, common to all is their integration with the concurrency control mechanism of the DBMS [119].

8.3 Oracle parallel cache management

This section describes the Oracle7 Parallel Database Server cache coherency policy as implemented on Goldrush and develops an analytical cache memory model for it.

8.3.1 Mechanism

In Oracle, locks obtained through the Goldrush DLM are used for the purpose of cache coherency management. For the purpose of concurrency control, conventional transaction locks are used. The owners of DLM locks are the Oracle instances, while transaction locks are owned by individual transactions.

The Oracle instance on each node maintains a local cache under the Least Recently Used (LRU) page replacement scheme, used to cache database pages. When a transaction on a particular instance requires access to a page, the instance acquires a DLM lock on the page on behalf of the transaction and reads the page into the cache.

The DLM locks associated with each database page in the cache are retained after the transaction that requested the page has committed, provided no transactions in other nodes are waiting for the same lock. A retained DLM lock is released or downgraded, and the associated cache copy of the database page may be purged (depending on the lock conflict - see below), if a DLM lock on the same page is subsequently requested on behalf of another transaction executing on a different node. A DLM lock may be requested on behalf of a remote transaction many times during the lifetime of a local transaction. A DLM lock may, therefore, be acquired and released

many times if the database page it covers is needed by transactions on other instances. For example:

1. Instance A becomes the owner of the DLM lock covering the data page containing row R1 and updates the row;
2. Instance B requests the page from instance A to update row R2;
3. Instance A releases the DLM lock;
4. Instance B becomes the owner of the page and the DLM lock and does its update of row R2;
5. Instance A requests the page from instance B to update row R3;
6. Instance B releases the page and the DLM lock;
7. Instance A becomes the owner of the page and DLM lock and updates row R3;
8. Instance A commits its transaction and still owns the DLM lock and the page until another instance requests the page.

The retained DLM lock is also released if the buffer copy of the page is pushed out of the cache. This happens when an instance must read in a new database page and the cache is full. In this situation one or more database pages from the bottom of the cache are written to disc to free the space required. This is also called a *foreground write*.

The DLM locks are of two types: shared (S) and exclusive (X). With shared DLM locks only shared read requests can be granted to a local transaction, and the share-locked page may be present in the cache of other instances. Exclusive DLM locks ensure that no other instance has a copy of the page and update requests can be granted locally. At transaction commit time, redo-log entries are written to disc to guarantee that the updates are permanently reflected in the database. The updated database pages, however, are not immediately propagated to disc (exclusive DLM locks are retained on these pages). If subsequent transactions under the same instance update the same blocks, only the final versions need to be propagated to disc, before releasing the DLM

X locks, thereby saving write I/O and improving overall system throughput. A DLM X lock is either downgraded to a DLM S lock or released if a DLM lock on the same database block is requested by a remote node either in S or X mode. When a DLM X lock is about to be released or downgraded, the (dirty) data block is written to disc. The requesting node can then read the updated database page, after the propagation to disc is completed.

The Oracle cache coherency policy that is to be captured in the model is as follows. Depending on the current status of the requested page and the requested lock mode, the following cases arise:

1. An instance requests a database page (for read only or update access) on behalf of a transaction and discovers that it is not present in any of the other instances' caches. In this case the page is read in from disc and the appropriate DLM lock is set (S or X).
2. An instance requests a database page for read only access on behalf of a transaction. The required page may be:
   (a) Held locally under DLM X lock. In this case no action is taken.
   (b) Held locally under DLM S lock. In this case no action is taken.
   (c) Held by a remote instance under DLM X lock. In this case the remote instance writes to disc the dirty page and downgrades its lock on this page from X to S. The requesting node's request for an S lock is granted after it reads the copy of the page from disc.
   (d) Held by one or more remote instances under DLM S lock. In this case the instance reads the page from disc and locks it in S mode.
3. An instance requires a database page for update access. Again, the required page may be:
   (a) Held locally under DLM X lock. In this case no action is taken.

(b) Held locally under DLM S lock. In this case, if one or more remote instances hold a copy of the page under DLM S lock, their locks are revoked. Then the requesting instance upgrades its lock on the page from S to X.

(c) Held by a remote instance under DLM X lock. In this case, the remote instance first writes the dirty database page to disc. The remote instance then releases its lock and the requesting instance acquires it.

(d) Held by one or more remote instances under DLM S lock. In this case all the S locks on the remote instances are revoked. The instance reads the page from disc and places a DLM X lock on the page.

Writes are also propagated to disc, if an updated page is pushed out of the cache (buffer flushing) due to the cache filling up. When a page is pushed out of the cache, its associated DLM lock (S or X) is also released.

8.3.2 Oracle cache model

The model is derived following the approach originally developed by Dan and Yu in [2]. The Oracle parallel cache management as described above is similar to the "Deferred until Transfer or Flushing" policy detailed in [2]. The model allows one to estimate the number of DLM X and S locks held on pages in a particular node's cache. This, in turn, allows one to obtain an estimation of local and remote cache hit probabilities.

Consider a fragment f located on disc d of node n. Denote it by (n,d,f). For each node m, suppose that:

$LocRds(m)$ is the number of reads required on the data of (n,d,f) by transactions local to m;

$RemRds(m,i)$ is the number of reads required on the data of (n,d,f) by remote to m transactions running on node i;

$TotRemRds(m)$ is the number of reads required on the data of (n,d,f) by remote to m transactions, $TotRemRds(m) = \sum_{i=1, i \neq m}^{N} RemRds(m,i)$.

Let

$bp(m,k)$ be the probability that the k^{th} buffer location from the top of the cache at node m contains a page of data from (n,d,f);

$xbp(m,k)$ be the probability that the page is from (n,d,f) and is holding an X lock;

$BufPgs(m,k)$ denote the average number of pages of data from (n,d,f) present in the top k buffer locations of the cache at node m;

$XBufPgs(m,k)$ and $SBufPgs(m,k)$ denote the average number of pages from (n,d,f) holding X and S locks, respectively, in the top k locations of the cache at node m.

Then

$$BufPgs(m,k) = \sum_{l=1}^{k} bp(m,l)$$

$$XBufPgs(m,k) = \sum_{l=1}^{k} xbp(m,l)$$

$$SBufPgs(m,k) = BufPgs(m,k) - XBufPgs(m,k).$$

A recursive procedure can be set up to determine $bp(m,k+1)$ and $xbp(m,k+1)$ for $k \geq 1$ given $bp(m,l)$ and $xbp(m,l)$ for $l = 1,\ldots,k$, as follows. Consider a smaller cache consisting of the top k locations only. Buffer location $(k+1)$ of the cache receives the block that is pushed down from location k. Suppose that:

$PgsPushed(m,k)$ is the number of pages from (n,d,f) that are pushed down from location k in the cache at node m caused by transactions running on node m;

$XPgsPushed(m,k)$ is the number of pages from (n,d,f) holding X locks that are pushed down from location k in the cache at node m caused by transactions running on node m;

$TotPgs$ is the size (in pages) of the data from (n,d,f);

$upd(m)$ is the probability that a page from (n,d,f) accessed by transactions running on node m is also updated.

The next two equations are based on the following assumption. Under steady-state conditions, in the long term the number of pages that get pushed down from the top k locations of the cache equals the number of pages that are brought into the top k locations. Hence:

$$PgsPushed(m,k) = LocRds(m)[1 - \frac{BufPgs(m,k)}{TotPgs}] - (\sum_{i=1,i\neq m}^{N} RemRds(m,i) \times upd(i)) \frac{BufPgs(m,k)}{TotPgs} \quad (8.1)$$

The first term on the RHS of the equation denotes the number of pages required by local transactions and not found in cache. Such pages are brought into cache from disc and placed under DLM X or S locks. The second term is the number of pages that are required by remote transactions for update and are found in instance m's cache (under an X or S lock). Such pages are taken out of the cache (after being written to disc).

$$XPgsPushed(m,k) = LocRds(m) \times upd(m)[1 - \frac{XBufPgs(m,k)}{TotPgs}] - TotRmRds(m) \frac{XBufPgs(m,k)}{TotPgs} \quad (8.2)$$

The first term on the RHS of the last equation denotes the number of pages required for update by local transactions and not found in cache. Such pages are brought into cache from disc and placed under a DLM X lock. The second term is the number of pages that are requested by remote transactions and are found to be under a

DLM X lock in instance m's cache. Such pages must have their DLM X lock revoked, either downgraded to S or set to "null".

The next two equations follow from another assumption: the expected value of finding a page from (n,d,f) in the $(k+1)^{th}$ buffer position of the cache of node m, $bp(m,k+1)$, is approximately the same as the probability of finding a page from (n,d,f) in the $(k+1)^{th}$ buffer position of the cache of node m in the event that a page is pushed down from location k to location $(k+1)$.

Therefore, the probability $bp(m,k+1)$ can be approximated as:

$$bp(m,k+1) \approx \frac{PgsPushed(m,k)}{\sum_{\forall(n,d,f)} PgsPushed(m,k)} \tag{8.3}$$

and

$$xbp(m,k+1) \approx \frac{XPgsPushed(m,k)}{PgsPushed(m,k)} bp(m,k+1) \tag{8.4}$$

Given that the next page accessed by a transaction at node m is from (n,d,f) let:

$h^{l,X}(m)$ be the probability that the page is found in the cache of node m under a DLM X lock;

$h^{r,X}(m)$ be the probability that the page is found in one of the remote caches under a DLM X lock;

$h^{l,S}(m)$ be the probability that the page is found in the cache of node m under a DLM S lock;

$h^{r,S}(m)$ be the probability that the page is found in one of the remote caches under a DLM S lock, and it is not found locally;

B be the size of the cache at each node.

The first two probabilities are easy to obtain, since a page under a DLM X lock can be in at most one instance's cache. Thus:

$$h^{l,X}(m) = \frac{XBufPgs(m,B)}{TotPgs} \tag{8.5}$$

$$h^{r,X}(m) = \frac{\sum_{i=1\ldots N,i\neq m} XBufPgs(i,B)}{TotPgs} \tag{8.6}$$

The sets of pages with DLM S and X locks are mutually exclusive, and therefore:

$$h^{l,S}(m) = \frac{SBufPgs(m,B)}{TotPgs} \tag{8.7}$$

Also:

$$h^{r,S}(m) = [1-(h^{l,X}(m)+h^{r,X}(m))][1-\frac{SBufPgs(m,B)}{TotPgs-\sum_{i=1\ldots N} XBufPgs(i,B)}] \tag{8.8}$$

$$\times[1-\prod_{j=1\ldots N,j\neq m}(1-\frac{SBufPgs(j,B)}{TotPgs-\sum_{i=1\ldots N} XBufPgs(i,B)})]$$

Here, the first term is the probability that the page does not lie in the set of pages under an X lock. The second term is the conditional probability that the page is not present in the local cache, given the first condition. The third term is the conditional probability that the page appears in one of the remote caches, given the first condition (the second condition has no effect on this term since the effects are independent).

Finally, the overall cache hit probability for a page of the data from (n, d, f), $h(m)$, can be written as:

$$h(m) = h^{l,X}(m)+h^{r,X}(m)+h^{l,S}(m)+h^{r,S}(m)$$

8.4 Informix cache management

8.4.1 Mechanism

As Informix uses the principle that data operations are always executed where data resides, there is no distributed lock management involved. Lock management is only at the local level of a co-server.

Within a co-server's shared memory, database buffers storing database pages are organised into LRU buffer queues. Informix uses two types of locks to manage access to shared-memory buffers: shared (S) and exclusive (X) locks, each of which enforces the required level of thread isolation during execution. A buffer is in shared mode if multiple threads have access to the buffer to read the data and none intends to modify the data. A buffer is in exclusive mode if a thread demands exclusive access to the buffer. All other threads requesting access to the buffer are placed on the waiting queue. When the executing thread is ready to release the exclusive lock, it wakes the next thread in the waiting queue.

The process of accessing a data buffer has the following steps [6]:

- Identify the requested data by physical page number;
- Determine the level of lock access – shared or exclusive;
- Try to locate the page in the memory. If it is not found, locate a free buffer and read the page. Otherwise, proceed with the next step.
- Test the lock-access level of the buffer. If it is compatible with that of the request, lock the buffer. Otherwise, wait in a queue for the buffer.
- After successful locking, process the buffer as required.
- When finished, release the lock and place the buffer on the most-recently used end of the queue.
- Wake waiting threads with compatible lock access types, if any exist.

8.4.2 Informix cache model

The model developed for Oracle may be adapted for the much simpler cache management of Informix. Unlike Oracle, in Informix only pages local to a node can be held in the node's cache. If a transaction requires a page from a node other than the one it is running on, the remote node has to bring the page into its own cache on behalf of

the transaction and transmit the data. Let n denote one of the nodes. Consider a fragment f located on disc d of node n, denoted by (d, f). Let:

LocRds be the number of reads required on the data from (d, f) by local transactions;

RemRds(i) be the number of reads required on the data from (d, f) by remote transactions running on node i;

Let

$bp(k)$ be the probability that the k^{th} buffer location from the top of the cache contains a page of data from (d, f);

$xbp(k)$ be the probability that the page is from (d, f) and is holding an X lock;

BufPgs(k) denote the average number of pages of data from (d, f) present in the top k buffer locations;

XBufPgs(k) and *SBufPgs*(k) denote the average number of pages from (d, f) holding X and S locks, respectively, in the top k locations of the cache.

As for Oracle:

$$BufPgs(k) = \sum_{l=1}^{k} bp(l)$$

$$XBufPgs(k) = \sum_{l=1}^{k} xbp(l)$$

$$SBufPgs(k) = BufPgs(k) - XBufPgs(k).$$

A similar recursive procedure can be used for $bp(k+1)$ and $xbp(k+1)$ for $k \geq 1$ given $bp(l)$ and $xbp(l)$ for $l = 1, \ldots, k$. If:

PgsPushed(k) is the number of pages from (d, f) that are pushed down from location k in the cache,

XPgsPushed(k) is the number of pages from (d, f) holding X locks that are pushed down from location k in the cache,

TotPgs is the size (in pages) of data from (d, f),

upd is the probability that a page from (d, f) accessed by local transactions is also updated,

upd(i) is the probability that a page from (d, f) accessed by remote transactions running on node i is also updated,

then the following steady-state equations can be used in place of (8.1) and (8.2):

$$PgsPushed(k) = LocRds[1 - \frac{BufPgs(k)}{TotPgs}] + \sum_{i=1, i \neq n}^{N} RemRds(i)[1 - \frac{BufPgs(k)}{TotPgs}] \quad (8.9)$$

The first term on the RHS of the equation denotes the number of pages required by local transactions and not found in the cache. Such pages are brought into cache. The second term is the number of pages that are required by remote transactions and are not found in the cache. Such pages are brought into cache on behalf of the remote transactions.

$$XPgsPushed(k) = LocRds \times upd[1 - \frac{XBufPgs(k)}{TotPgs}] + \sum_{i=1, i \neq n}^{N} RemRds(i) \times upd(i)[1 - \frac{XBufPgs(k)}{TotPgs}] \quad (8.10)$$

The first term on the RHS of the last equation denotes the number of pages required for update by local transactions and not found in the cache. Such pages are brought into the cache from disc and placed under an X lock. Similarly, the second term is the number of pages that are requested by remote transactions for update and are not found to be in the cache.

As in the Oracle model, the probability $bp(k+1)$ can be approximated as:

$$bp(k+1) \approx \frac{PgsPushed(k)}{\sum_{\forall (d,f)} PgsPushed(k)} \quad (8.11)$$

and

$$xbp(k+1) \approx \frac{XPgsPushed(k)}{PgsPushed(k)} bp(k+1) \quad (8.12)$$

Given that the next page accessed by a transaction at node n is from (d, f), let:

h^x be the probability that the page is found in the cache under an X lock;

h^s be the probability that the page is found in the cache under an S lock;

The two probabilities are easy to obtain:

$$h^x = \frac{XBufPgs(B)}{TotPgs} \tag{8.13}$$

$$h^s = \frac{SBufPgs(B)}{TotPgs} \tag{8.14}$$

In (8.13) and (8.14) B is the size of the cache (in pages) of each node. Finally, the overall cache hit probability for a page from (d, f) can be written as:

$$h = h^x + h^s$$

8.5 Implementing the cache models

The models described above have been implemented within STEADY. DPTool (see Section 3.4) estimates the number of read (R_m) and write (W_m) operations required by the transactions of the given application running on node m to data pages from node n, disc d, and fragment f, for each n, d, and f. This is done through analysing the query trees and data placement. Since each node runs the same application, the estimated number of reads is the same for all values of m; the same applies to the number of writes. Therefore, R_m and W_m may be written more simply as R and W. For a given node n, disc d, and fragment f, the input parameters for the Oracle model are available as follows:

$$LocRds(m) = RemRds(m,i) = R \text{ and } upd(m) = \frac{W}{R}.$$

Similarly, for Informix, they are:

$$LocRds = RemRds(i) = R \text{ and } upd = upd(i) = \frac{W}{R}.$$

The size in pages of data from fragment f on disc d of node n, $TotPgs$, is also readily available for both models from the data placement.

The Oracle cache model is implemented as a loop of B iterations, where B is the size in pages of the cache. Initially, $BufPgs(m,0)$ and $XBufPgs(m,0)$ are set to 0. In the body of the loop, on iteration k, $PgsPushed(m,k)$ and $XPgsPushed(m,k)$ are computed for each fragment f, disc d and node n, according to equations (8.1) and (8.2). Once this is done, $bp(m,k+1)$ and $xbp(m,k+1)$ can be calculated from equations (8.3) and (8.4). They are added to $BufPgs(m,k)$ and $XBufPgs(m,k)$, respectively, to give $BufPgs(m,k+1)$ and $XBufPgs(m,k+1)$. The loop body is then repeated. After this loop, the four cache hit probabilities $h^{l,x}(m)$, $h^{r,x}(m)$, $h^{l,s}(m)$ and $h^{r,s}(m)$ are computed according to equations (8.5), (8.6), (8.7) and (8.8), respectively. Note that since all nodes are assumed to have equal cache sizes, run the same transactions, and have physical accessibility to data from any node, disc and fragment of the system, the cache hit probabilities are the same for all nodes. The loop producing the values of the probabilities for node m effectively produces these values for every other node. The run time complexity of the algorithm is therefore $O(B)$.

In the case of Informix, however, a node only has accessibility to local data. Since in general local data differs from node to node in accordance with the data placement, each node will have unique cache hit probabilities. Correspondingly, a loop of B iterations is executed for each node. Again, $BufPgs(0)$ and $XBufPgs(0)$ are initially set to 0. In the body of each loop, on iteration k, $PgsPushed(k)$ and $XPgsPushed(k)$ are computed for each fragment f and disc d, according to (8.9) and (8.10). Once this is done, $bp(k+1)$ and $xbp(k+1)$ can be calculated from (8.11) and (8.12). They are added to $BufPgs(k)$ and $XBufPgs(k)$, respectively, to give $BufPgs(k+1)$ and $XBufPgs(k+1)$. The loop body is then repeated. After each loop, the two probabilities h^x and h^s are computed according to (8.13) and (8.14), respectively. The run time complexity of this algorithm is $O(N \times B)$, where N is the number of nodes.

8.6 Discussion and results

Some examples of the predictions of the Oracle model are presented to illustrate the model. Several experiments are performed. For the purpose, the tables *Warehouse*, *District*, *Customer* and *New-Order* from the TPC-C benchmark are used. For each warehouse there are 10 districts, each of which has 1000 customers; 100 new orders per warehouse are assumed. For the experiments, the database is allowed to scale by increasing the number of warehouses and changing the other tables' sizes appropriately. Each table is then fragmented into w fragments of equal size, where w is the number of warehouses in the database. All fragments are placed on the nodes of the machine.

In addition, experiments are performed with increasing the number of nodes while keeping the database size (number of fragments) constant. For a database composed of 20 warehouses, the number of nodes is increased from 2 to 16.

For all experiments each node runs a simple application consisting of two transactions. The first is modelled on the Payment transaction from TPC-C. It takes as input a warehouse, district and customer id as well as a payment amount, and updates the relevant rows of the *Warehouse*, *District* and *Customer* tables. Physically, a single database page of each table is read into cache and updated. Following this, the entire *New-Order* table is scanned. The input parameters for each transaction are chosen at random and therefore a random page from each table is updated each time. Thus, each page is equally likely to be updated. The second transaction represents a decision support query requiring a scan of the whole of the *Customer* table. This transaction is much less frequent than the first one (ratio of 10,000:1). However, it requests a shared lock on every *Customer* page.

8.6.1 A simple experiment

Consider first the results from a simple experiment where a 20-warehouse database is created for a 10-node configuration. Each node has a cache size of 5000

pages. The results are presented in Table 8-1. The fragment size in tuples and pages is given in column 3. The probabilities indicate that all pages from the fragments of the small and frequently accessed tables *Warehouse*, *District* and *New-Order* are held entirely in memory. An X-locked page can be held at only one node's cache at any one time. Thus the pages from *Warehouse* and *District* fragments are evenly divided between the 10 node's caches. S-locked pages, on the other hand, can occupy any number of caches and therefore the pages of *New-Order* (which are only scanned) are present in every node's cache. Since table *Customer* is both read and updated, pages from its fragments can be held under X or S locks, locally or remotely. *Customer* pages are more likely to be on disc than in cache, as the last column indicates.

Relation	Fragments	Fragment size in tuples and pages	Cache hit probabilities for a page from a fragment for each relation				
			$h^{l,X}$	$h^{l,S}$	$h^{r,X}$	$h^{r,S}$	h
Warehouse	20	1 (1)	0.1	0	0.9	0	1
District	20	10 (2)	0.1	0	0.9	0	1
Customer	20	10,000 (5209)	0.0209	0.0265	0.1881	0.2016	0.4371
New-Order	20	100 (3)	0	1	0	0	1

Table 8-1 Cache hit probabilities

8.6.2 Varying the number of nodes

Consider the way the cache hit probabilities for table *Customer* change with an increase in the number of nodes, while keeping the same database size of 20 warehouses. The results are presented in Table 8-2. Consider first $h^{l,X}$, which decreases with the increase in the number of nodes. This follows since pages may be X-locked by at most one instance, and the more nodes there are, the smaller the chance that any particular one of them will be the exclusive owner of a page. Correspondingly, the probability that a page is found on a remote node ($h^{r,X}$) increases. The values of $h^{l,S}$ and $h^{r,S}$ increase with the number of nodes. This follows, since the more nodes there are, the larger the overall available cache space, with decreasing proportion of it occupied by X-locked pages. Hence the more room for S-locked pages and the higher

probability of finding a page under an S lock. Note that S-locked pages may be in more than one cache at the same time.

Number of nodes	$h^{l,X}$	$h^{l,S}$	$h^{r,X}$	$h^{r,S}$	h
2	0.0231	0.0240	0.0231	0.0234	0.0937
3	0.0228	0.0244	0.0457	0.0469	0.1398
4	0.0226	0.0247	0.0677	0.0701	0.1851
5	0.0223	0.0250	0.0892	0.0931	0.2297
6	0.0220	0.0253	0.1102	0.1158	0.2733
7	0.0218	0.0256	0.1305	0.1381	0.3159
8	0.0215	0.0259	0.1503	0.1599	0.3575
9	0.0212	0.0262	0.1695	0.1811	0.3979
10	0.0209	0.0265	0.1881	0.2016	0.4371
11	0.0206	0.0267	0.2062	0.2215	0.4751
12	0.0203	0.0270	0.2237	0.2406	0.5117
13	0.0200	0.0273	0.2406	0.2589	0.5468
14	0.0198	0.0276	0.2569	0.2763	0.5806
15	0.0195	0.0279	0.2727	0.2927	0.6128
16	0.0192	0.0282	0.2879	0.3082	0.6435

Table 8-2 Cache hit probabilities for different number of nodes

Figure 8-1 shows the change in probabilities for table *Warehouse* with the increase in number of nodes. The pages of this table are to be found entirely within the caches of the nodes. With the increase in the number of nodes, the smaller the probability that a page is found in a particular node becomes, while the probability that a remote node's cache contains the page grows.

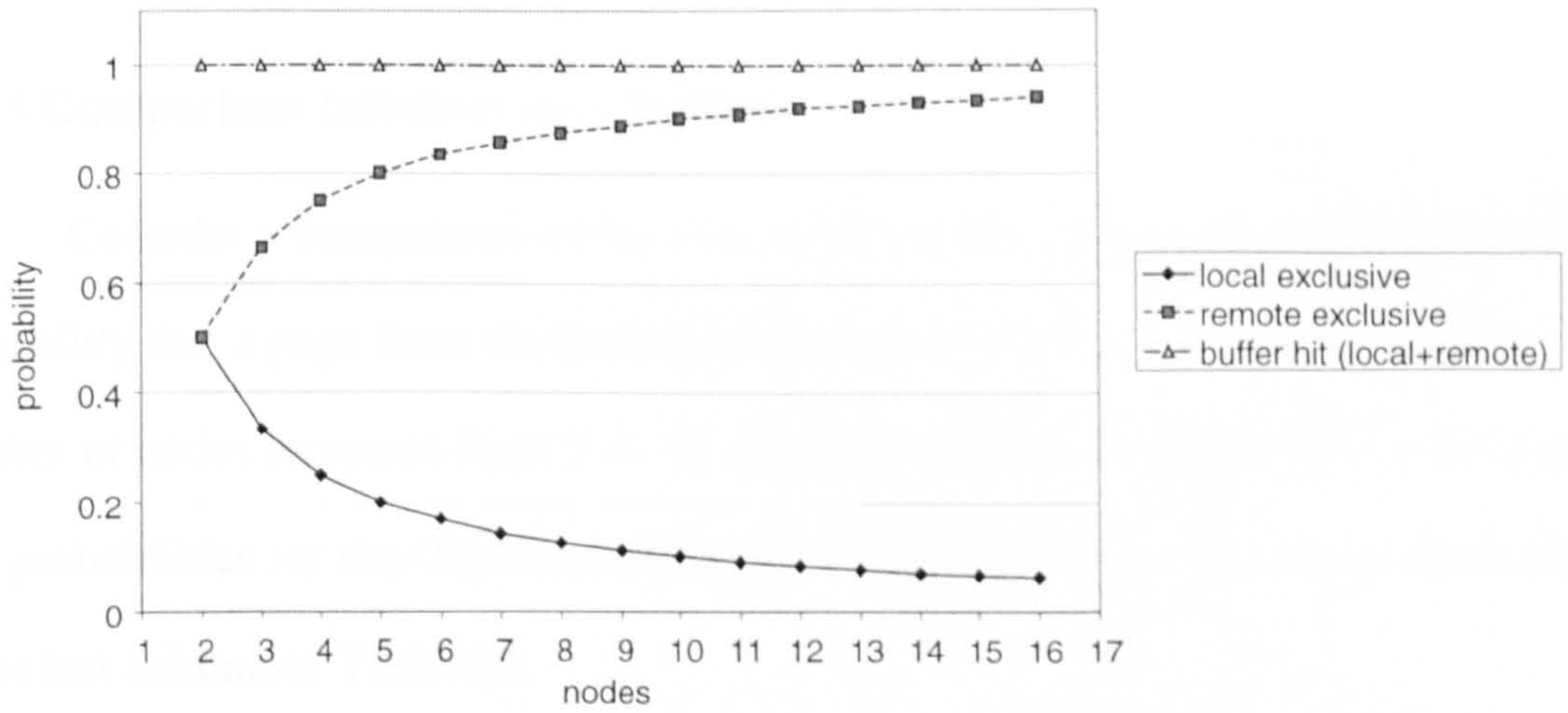


Figure 8-1 Probability that a *Warehouse* page is held under X lock

8.6.3 Varying the number of warehouses

Consider now the variations in probabilities as database size increases. The number of nodes is set to 10, while the number of warehouses increases from 20 to 100 in steps of 5. The four probabilities for table *Customer* are shown in Figure 8-2. All four probabilities decrease smoothly with the increase in the number of warehouses. This is because such an increase implies an increase in the size of all database relations. Since the cache size remains fixed, this implies that a decreasing proportion of the data will be held in cache, in either exclusive or shared mode, locally or remotely.

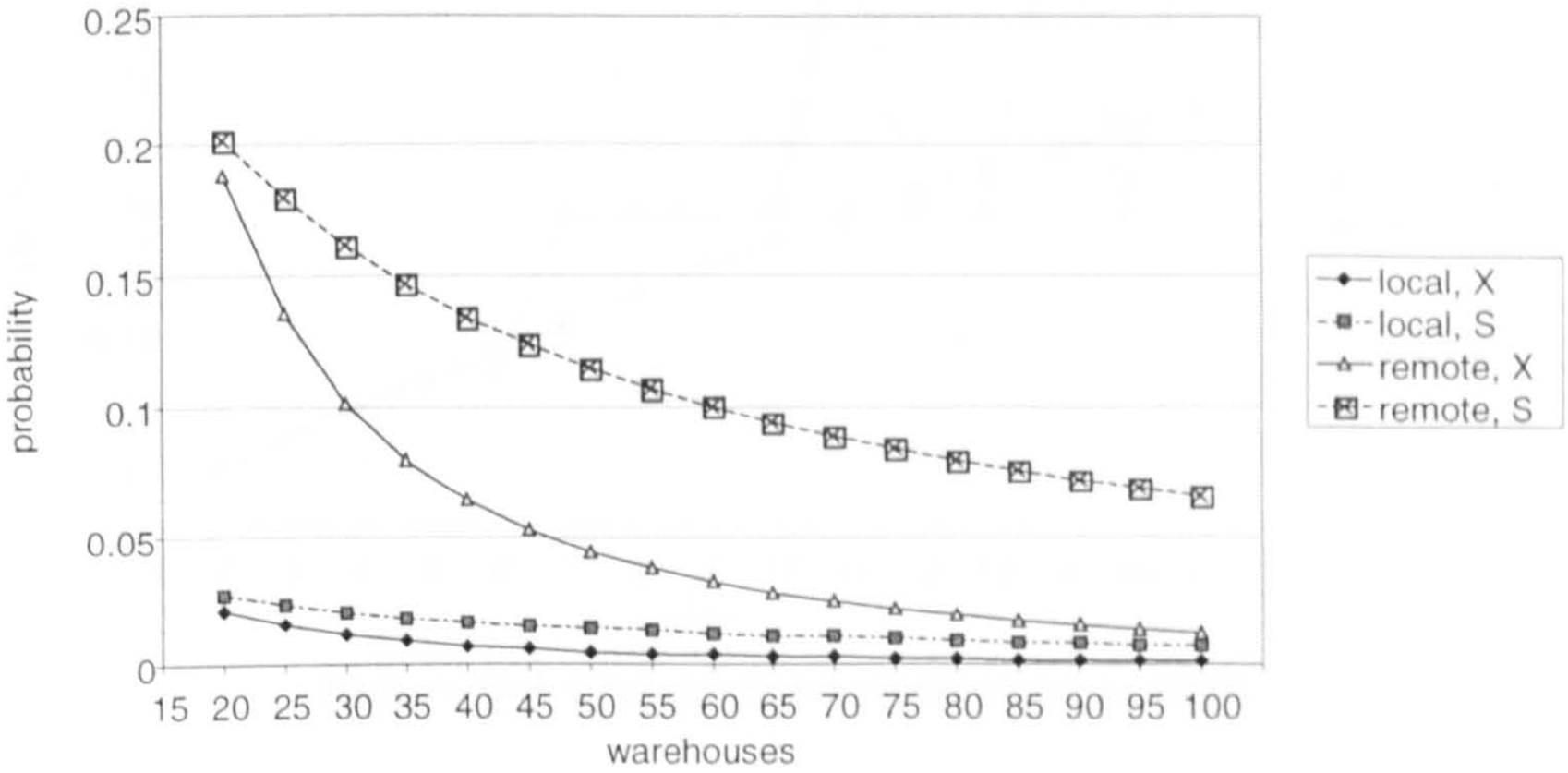


Figure 8-2 Probability that a *Customer* page is in cache

8.6.4 Comparison between two models

Consider a comparison of the two cache models. Figure 8-3 shows the overall probability that a page from the *Customer* relation is found in the cache of node 1 as the number of nodes increases from 2 to 16, while the number of warehouses is fixed at 20. The probabilities for the Oracle model shown in the figure are the same as those shown in the last column of Table 8-2.

The discontinuity in the Informix cache hit probability is due to the effect of data placement. Unlike Oracle, where node 1 accesses and stores in its cache pages from each of the 20 *Customer* fragments, in Informix node 1 may only access local fragments. For each distinct number of nodes, the data placement strategy assigns a

potentially different number of fragments to node 1. For example, in the case of 7, 8, 9 and 10 nodes, node 1 is assigned two *Customer* fragments, while in the case of 11, 14 and 16 nodes, node 1 receives a single fragment. The larger the amount of pages that need to be held in cache, the smaller the probability that any one of them will be found in cache. Thus, the cache hit probability for a *Customer* page when a single customer fragment is accessed (around 0.95 on the graph) is twice as large as the case when the pages from two fragments are accessed (around 0.5).

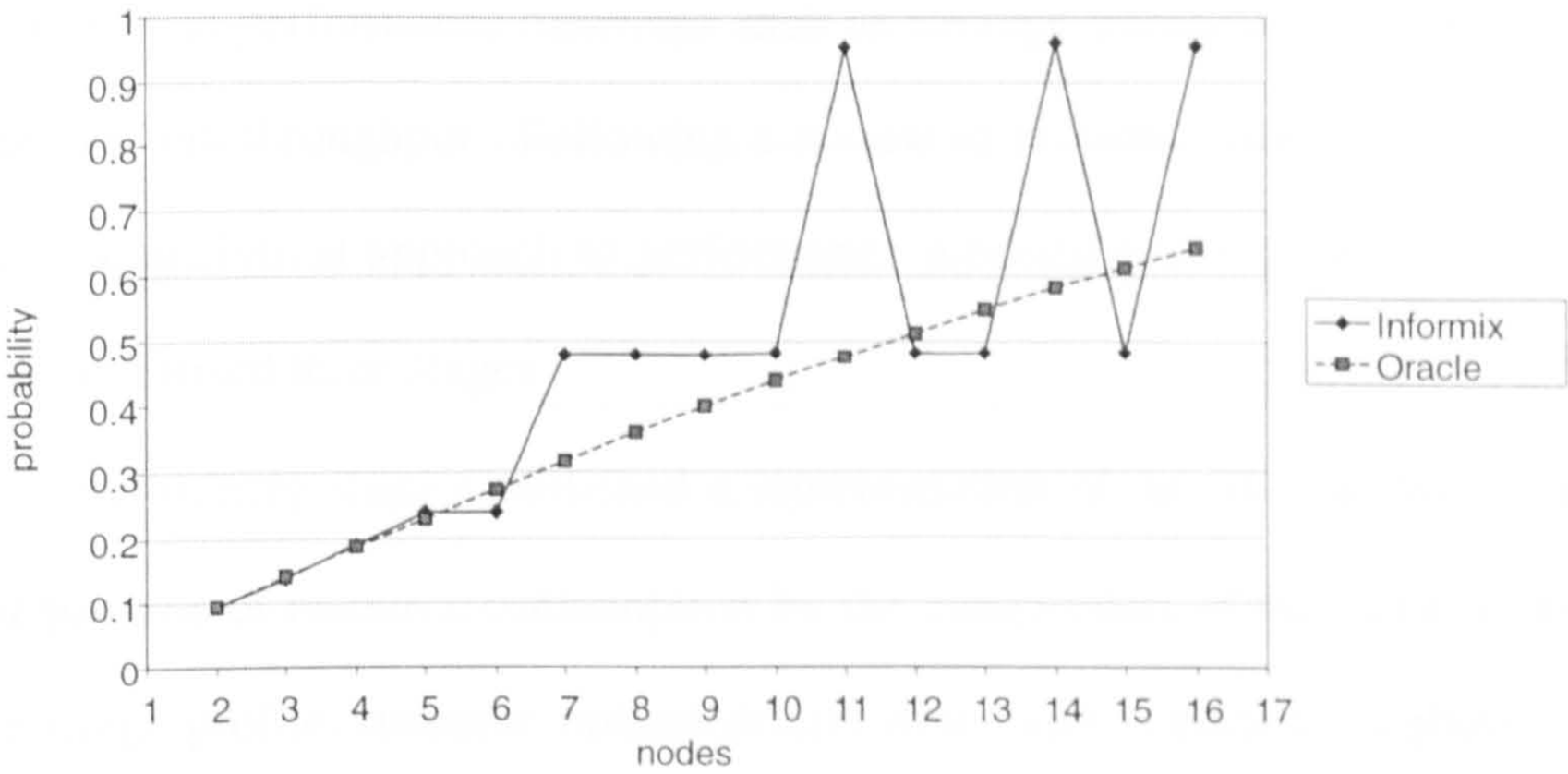


Figure 8-3 Comparison of the two models

Chapter 9

CONCLUSION

9.1 Review of thesis and conclusions

The thesis has investigated the modelling of parallel relational database systems in order to obtain performance measures such as average transaction response time and maximum system throughput. Following a review of relevant issues in Chapters 2 and 3, a practical analytical approach to performance modelling was proposed. This novel approach comprised three stages.

A preliminary stage established a representation of parallel database activity in terms of patterns of resource consumption by the components of the system. From the resource usage profile, resource utilisation and maximum system throughput could be computed easily. The preliminary stage was the subject of Chapter 4.

The second and third stages of the approach were concerned with response time estimation of transactions. Resource usage profiles were mapped to open, multi-class queueing networks. The obtained networks were not in product form, due to both non-exponential service times at queueing centres, and synchronisation between query execution phases (pipelines, partitioned parallelism) which imposed synchronisation on the transitions between queues. A two-step solution was proposed to tackle this problem. First, queue waiting time estimates for individual resources were obtained without considering synchronisation issues. Second, using these estimates, the transaction response time was approximated, taking into account synchronisation and dependencies between query execution phases.

The second stage of the approach implemented the first of the above two steps and was the subject of Chapter 5. The waiting times at queueing network centres were

estimated given their involvement in database activity as specified by the resource usage profile. Due to the non-exponential service times at queueing centres, exact product-form solutions were not applicable. A variety of approximation techniques, some quite complex, could be used to estimate queue waiting time. To choose the most appropriate one, a study was conducted in which a selection of techniques was applied to a large number of examples. None of the techniques was found to perform better than the rest in all cases and a heuristic rule was derived, which performed better than the other techniques. The rule performs consistently well over a wide range of examples without being computationally expensive to apply.

The third stage of the approach estimated the average transaction response time. Given the estimated resource waiting times and the original resource usage profile, response time was obtained through a traversal of the profile. Account was taken of different types of synchronisation mechanisms, such as relational pipelines between two or more operators of the execution schedule, partitioned parallelism within a single operator, and blocking. This was presented in Chapter 6.

The proposed performance estimation approach has been implemented as part of a stand-alone tool for rapid performance prediction. The approach was validated through comparisons between estimated (by the tool) and actual performance of Informix XPS queries executed on an ICL Goldrush platform. The procedure and the results obtained were the subject of Chapter 7.

Other types of database behaviour, such as background processing or cache maintenance can be taken into account within the framework of the method. To show this, in Chapter 8 models of the cache management mechanisms of Oracle and Informix were developed to predict cache hit probabilities that can be incorporated into the resource usage profile and thus taken into account in further performance estimation stages.

The overall conclusion that can be drawn from this work is that the developed analytical method for performance estimation is valid and produces results, which are within an acceptable level of error. In particular, the average errors computed for transaction throughput and average response time in Chapter 7 (15% and 20%, respectively) are very similar to those the tools discussed in Section 3.3.3 aim to achieve. Thus, for the types of queries for which it has been validated, the proposed analytical method is as good as the currently available commercial products. At the same time, the approach is not computationally demanding and is relatively simple. It is thus plausible to implement it as the core of a stand-alone tool for rapid performance estimation to aid in tasks such as capacity planning and data placement.

9.2 Further work

Some directions for further work and improvement of the modelling methodology are discussed briefly in this section.

Heuristic rule. The heuristic rule proposed in Chapter 5 is developed in the context of queueing networks derived from a representation of database transactions. In addition to improving the rule within this context, e.g. finding better weighting for the utilisation and relative visit ratio, the heuristic may prove applicable to queueing networks that arise in other domains. This needs to be established through further experimentation. The heuristic is most likely to work well with networks with high proportion of feedback traffic and relatively low variability in service time.

Cache models validation. Although the principles behind the cache models have been extensively validated by simulation in [1, 2], they remain to be validated against actual system measurements. The first step of the validation process would be to compare predicted and measured cache hit probabilities of different types and for different tables. To obtain reliable measurements, a suite of transactions that fetch database pages in a random but controlled manner must be devised. The transactions

would run on the nodes of the machine over a long period of time to allow each cache to reach a steady state. The cache hit probabilities for the systems can then be obtained either through the performance monitoring tools of the DBMS engine, or through a calculation based on the observed state of the cache. The second step of the validation process would be to compare the measured and predicted response time (and throughput) of transactions, which fetch some of their pages from disc and others from cache. The cache hit probability would be integrated within the task blocks of the transaction. Any discrepancy at this stage would indicate that the system performs additional activity, not represented in the models. This may lead to further refinements of the cache models.

Concurrency control. Currently, the performance prediction approach does not take in to consideration concurrency control issues. For the validation of the approach in Chapter 7 it is assumed that no lock contention occurs. There are a number of studies in the literature [120, 121, 122] that focus on concurrency control issues. Based on such studies and a sound understanding of the concurrency control mechanism of the modelled DBMSs, an analytical method for prediction of the average lock waiting time for different types of locks for different transactions may be formulated. This quantity may then inserted at an appropriate place into the task/resource blocks as a basic operation that does not require service from a physical resource, but consumes time.

Background activity. Further investigation is needed in order to incorporate the effects of background activity such as flushing of buffers, checkpoints, logging etc. For example, periodic checkpoints may be modelled with simple task blocks consisting of specific basic cost items (e.g. *sequential_write*), chosen to represent such activity. Based on the policy, according to which the DBMS performs a checkpoint, the frequency of such task blocks may be determined. Models of background activity need to be validated with carefully chosen experiments.

Load balancing issues. The modelling methodology presented in the thesis does not consider load balancing explicitly. The load balancing mechanism of the DBMS partly affects the allocation of tasks to processors. For example, tasks may be sorted according to some criteria, and then allocated in descending order of cost to processors in order to achieve a balanced system. The allocation strategy may be static, where all tasks are allocated prior to the execution and each node knows what it has to execute [73] or adaptive, in which case each node will acquire the next task only when the current one is done [74]. The task/resource block formalism will need to be extended in order to capture query execution governed by a load balancing mechanism.

Other systems and architectures. The modelling methodology has only been validated against Informix XPS. Although the approach has been developed to model Oracle Parallel Server, it remains to be validated against real measurements from an Oracle/Goldrush system. In addition, the ability to represent other systems/architectures within the framework of model needs to be investigated. For example, the ICL Goldrush MegaServer platform is now being replaced by the new Trimetra Xtraserver architecture [123]. This is a NUMA system (see Section 2.2.4) and is likely to be more difficult to model. On the other hand, recent NUMA systems are proving successful as high-performance database servers, and it is therefore important to be able to model their performance.

Appendix A

This appendix presents the execution plan as produced by the Informix XPS 'explain plan' command. Shown are: the SQL text of the query, the chosen methods for executing the relational operations (SEQUENTIAL SCAN, HASH JOIN), and the execution plan itself. Partitioned parallelism is indicated in the 'width' column, which specifies the number of nodes involved in the operator.

```
QUERY:
------
select max(un80.entier) , count(*)
from un80, un30k, un60k
where un80.key = un30k.key
    and un80.key = un60k.key

Estimated Cost: 7
Estimated # of Rows Returned: 1

1)informix.un80: SEQUENTIAL SCAN

2)informix.un60k: SEQUENTIAL SCAN

DYNAMIC HASH JOIN (Build Outer)
    Dynamic Hash Filters: informix.un80.key = informix.un60k.key

3) informix.un30k: SEQUENTIAL SCAN

DYNAMIC HASH JOIN (Build Outer)
    Dynamic Hash Filters: informix.un80.key = informix.un30k.key

# of Secondary Threads = 28

XMP Query Plan

oper    segid   brid    width
----    -----   ----    -----
scan    7       0       1
scan    6       0       1
hjoin   4       0       8
scan    5       0       1
hjoin   3       0       8
group   2       0       8
group   1       0       1
```

Details of the way each of the 7 operators was executed are also produced. The number of rows consumed, produced, probed, built, etc., by each co-server are specified.

```
XMP Query Statistics

Cosvr_ID: 1
Plan_ID: 2510

type    segid   brid    information
----    -----   ----    -----------
scan    7   0   inst    cosvr   rows_prod   rows_scan
                ----    -----   ---------   ---------
                0       7       80          80
                ----    -----   ---------   ---------
                        1       80          80


scan    6   0   inst    cosvr   rows_prod   rows_scan
                ----    -----   ---------   ---------
                0       7       60000       60000
                ----    -----   ---------   ---------
                        1       60000       60000

hjoin   4   0   inst    cosvr   rows_prod   rows_bld    rows_probe  mem ovfl
                ----    -----   ---------   --------    ----------  --- ----
                0       1       13          13          7518        96  0
                1       2       12          12          7435        96  0
                2       3       8           8           7509        64  0
                3       4       8           8           7612        64  0
                4       5       9           9           7501        72  0
                5       6       5           5           7515        40  0
                6       7       10          10          7525        80  0
                7       8       15          15          7385        112 0
                        -----   ---------   --------    ----------  --- ----
                        8       80          80          60000       (14848)

scan    5   0   inst    cosvr   rows_prod   rows_scan
                ----    -----   ---------   ---------
                0       7       30000       30000
                ----    -----   ---------   ---------
                        1       30000       30000
```

```
hjoin   3    0    inst    cosvr   rows_prod  rows_bld   rows_probe mem ovfl
                  ----    -----   ---------  --------   ---------- --- ----
                  0       1       13         13         3785       96  0
                  1       2       12         12         3768       96  0
                  2       3       8          8          3743       64  0
                  3       4       8          8          3606       64  0
                  4       5       9          9          3761       72  0
                  5       6       5          5          3828       40  0
                  6       7       10         10         3780       80  0
                  7       8       15         15         3729       112 0
                  ----    -----   ---------  --------   ---------- --- ----
                          8       80         80         30000      (14848)
```

```
group   2    0    inst    cosvr   rows_prod  rows_cons  mem ovfl
                  ----    -----   ---------  ---------  --- ----
                  0       1       1          15         -1  -1
                  1       2       1          13         -1  -1
                  2       3       1          12         -1  -1
                  3       4       1          8          -1  -1
                  4       5       1          8          -1  -1
                  5       6       1          9          -1  -1
                  6       7       1          5          -1  -1
                  7       8       1          10         -1  -1
                  ----    -----   ---------  ---------  --- ----
                          8       8          80         (-1)
```

```
group   1    0    inst    cosvr   rows_prod  rows_cons  mem ovfl
                  ----    -----   ---------  ---------  --- ----
                  0       1       1          8          -1  -1
                  ----    -----   ---------  ---------  --- ----
                          1       1          8          (-1)
```

Appendix B

Tables B-1, B-2, B-3, and B-4 present the results of various approximation methods for queueing networks with non-exponential service times. The methods and the example they are applied to are discussed in Chapter 5. The tables present the percentage difference between transaction response times obtained by simulation and by each of the methods. The largest value for each experiment is underlined. Results for a range of bottleneck resource utilisations (40%, 50%, 60%, 70% and 80%) are presented.

id.	method	utilisation of bottleneck resource									
		T_1					T_2				
		0.4	0.5	0.6	0.7	0.8	0.4	0.5	0.6	0.7	0.8
01	M/G/1	-1.55	-3.68	-6.27	-10.51	-18.39	-7.09	-12.51	-17.02	-22.83	-30.67
	max. ent.	-9.46	-14.80	-20.88	-28.64	-39.43	-16.11	-24.77	-32.72	-41.68	-51.79
	decomp.	-3.91	-7.89	-13.04	-20.57	-32.24	-10.11	-17.65	-25.00	-34.16	-45.50
	mean val.	14.75	17.23	19.55	20.28	16.71	9.04	7.73	7.70	6.2	2.05
02	M/G/1	-1.8	-0.88	-1.94	-3.04	-2.08	-0.8	0.25	-0.7	-2.82	-1.58
	max. ent.	-0.5	0.89	0.25	-0.47	0.84	0.79	2.41	1.97	0.23	1.78
	decomp.	-1.42	-0.22	-0.95	-1.65	-0.23	-0.28	1.12	0.61	-1.07	0.67
	mean val.	-3.79	-3.53	-5.30	-7.28	-7.62	-3.85	-3.72	-5.59	-8.67	-8.76
03	M/G/1	39.41	43.74	44.43	38.41	22.72	-8.82	-11.6	-15.72	-22.21	-31.07
	max. ent.	34.75	37.08	35.63	27.62	10.49	-13.96	-19.11	-25.78	-34.56	-44.91
	decomp.	38.31	41.9	41.71	34.85	18.7	-10.17	-13.86	-19.01	-26.4	-35.6
	mean val.	6.15	6.15	5.46	2.75	-2.82	7.01	8.85	9.39	7.06	1.46
04	M/G/1	-4.52	-5.34	-9.13	-14.26	-19.39	-10.7	-13.64	-19.08	-26.44	-30.24
	max. ent.	-10.33	-13.61	-19.82	-27.32	-34.95	-17.96	-23.66	-31.58	-40.86	-46.93
	decomp.	-6.19	-8.36	-13.92	-21.30	-29.41	-13.03	-17.69	-25.21	-34.84	-41.66
	mean val.	6.49	9.29	9.21	8.20	8.44	3.31	4.45	2.84	-1.10	0.26
05	M/G/1	-2.55	-4.10	-6.86	-10.42	-17.33	-6.25	-9.68	-14.97	-20.03	-27.08
	max. ent.	-9.52	-13.90	-19.63	-26.28	-35.81	-14.32	-20.84	-29.07	-37.02	-46.23
	decomp.	-4.63	-7.81	-12.76	-19.20	-29.47	-8.93	-14.35	-22.12	-30.22	-40.5
	mean val.	11.81	14.32	15.71	16.47	13.35	8.14	8.66	7.10	5.96	2.33
06	M/G/1	-3.73	-6.19	-6.34	-9.37	-19.43	-6.77	-10.74	-12.20	-18.60	-29.78
	max. ent.	-11.42	-16.92	-20.75	-27.40	-39.70	-15.72	-23.07	-28.49	-37.98	-50.46
	decomp.	-5.91	-10.03	-12.67	-18.86	-32.12	-9.59	-15.63	-20.03	-29.65	-43.60
	mean val.	11.41	13.07	17.90	19.67	12.40	8.25	8.31	11.76	9.22	-0.02
07	M/G/1	-3.90	-7.32	-8.45	-16.85	-18.26	-8.83	-10.20	-16.53	-26.00	-24.21
	max. ent.	-11.40	-17.63	-22.09	-32.85	-38.23	-17.28	-22.09	-31.31	-42.81	-45.70
	decomp.	-5.46	-10.01	-12.73	-22.69	-26.53	-10.83	-13.70	-21.70	-32.75	-33.83
	mean val.	8.69	8.62	11.62	6.20	10.80	2.20	4.46	1.40	-5.26	3.84
08	M/G/1	0.73	1.27	0.72	-2.25	-8.12	4.03	2.67	-2.30	-7.44	-14.78
	max. ent.	-10.56	-15.21	-21.80	-31.37	-44.15	-9.69	-17.03	-27.99	-39.54	-52.80
	decomp.	-3.28	-6.12	-11.54	-21.02	-35.10	-1.39	-7.03	-17.46	-29.51	-44.64
	mean val.	20.44	27.52	34.19	38.70	40.19	26.42	32.19	33.86	35.68	34.47
09	M/G/1	-0.89	-1.36	-3.34	-7.05	-14.79	-3.26	-6.17	-12.42	-18.19	-26.46
	max. ent.	-9.74	-14.03	-20.14	-28.12	-39.43	-13.61	-20.67	-30.74	-40.36	-51.38
	decomp.	-3.67	-6.38	-11.45	-19.15	-31.41	-6.88	-12.51	-22.08	-31.92	-44.31
	mean val.	16.20	20.96	24.40	26.18	23.05	14.40	16.73	14.97	13.91	9.38

Table B-1 Percentage difference between results of methods and results of simulation.

id.	method	utilisation of bottleneck resource				
		0.4	0.5	0.6	0.7	0.8
11	M/G/1	2.49	2.24	1.94	6.39	6.06
	max. ent.	2.38	2.06	1.69	6.06	5.75
	decomp.	2.46	2.19	1.89	6.38	6.22
	mean val.	16.30	20.45	25.20	37.06	44.82
12	M/G/1	1.68	-0.77	-1.14	-2.59	-0.78
	max. ent.	1.35	-1.25	-1.81	-3.42	-1.68
	decomp.	1.60	-0.90	-1.33	-2.79	-0.88
	mean val.	20.45	23.84	30.81	37.92	52.47
13	M/G/1	0.42	-1.27	1.68	-2.28	1.50
	max. ent.	0.12	-1.71	1.06	-3.02	0.70
	decomp.	0.35	-1.39	1.52	-2.44	1.44
	mean val.	18.14	22.10	32.98	36.31	53.02
14	M/G/1	4.31	2.62	-0.72	-0.63	-3.18
	max. ent.	3.72	1.76	-1.82	-1.98	-4.57
	decomp.	4.15	2.34	-1.12	-1.15	-3.68
	mean val.	23.71	28.16	31.36	40.50	48.32
15	M/G/1	0.10	0.30	1.88	-3.66	-9.82
	max. ent.	0.48	0.79	2.44	-3.13	-9.37
	decomp.	0.26	0.60	2.40	-2.88	-8.75
	mean val.	12.32	17.02	24.56	25.00	26.78
16	M/G/1	2.23	0.29	1.17	-1.70	-1.86
	max. ent.	-0.60	-3.87	-4.60	-8.92	-10.56
	decomp.	1.34	-1.35	-1.56	-5.72	-7.48
	mean val.	24.92	29.96	39.73	45.64	57.43
17	M/G/1	-2.51	-7.41	-5.86	-17.46	-13.75
	max. ent.	-2.91	-8.00	-6.74	-18.52	-15.23
	decomp.	-2.61	-7.60	-6.20	-17.94	-14.52
	mean val.	19.64	20.75	31.16	23.59	39.81
18	M/G/1	4.74	3.55	1.80	-1.97	-4.94
	max. ent.	5.43	4.52	3.03	-0.54	-3.30
	decomp.	4.94	3.92	2.41	-1.05	-3.57
	mean val.	19.35	22.95	26.75	29.25	35.02
19	M/G/1	0.51	-2.66	-4.56	-9.60	-18.36
	max. ent.	0.41	-2.88	-4.99	-10.34	-19.46
	decomp.	0.55	-2.59	-4.43	-9.40	-18.06
	mean val.	7.16	6.66	8.51	8.29	5.90

Table B-2 Percentage difference between results of methods and results of simulation

tx.	method	utilisation of bottleneck resource				
		0.4	0.5	0.6	0.7	0.8
T_1	M/G/1	1.62	0.56	-1.61	-5.71	-13.18
	max. ent.	-1.53	-4.12	-7.94	-13.65	-22.34
	decomp.	0.66	-1.24	-4.56	-10.10	-19.15
	mean val.	13.78	15.91	16.85	15.49	9.85
T_2	M/G/1	-5.13	-8.02	-12.12	-17.05	23.76
	max. ent.	-6.54	-10.17	-15.17	-21.14	-29.01
	decomp.	-5.61	-8.93	-13.70	-19.56	-27.54
	mean val.	1.58	0.72	-1.26	-3.89	-8.20
T_3	M/G/1	-4.26	-6.80	-10.25	-15.24	-22.33
	max. ent.	-7.72	-11.73	-16.65	-22.94	-30.91
	decomp.	-5.44	-8.91	-13.56	-19.94	-28.42
	mean val.	9.60	10.10	9.33	6.39	0.40

Table B-3 Percentage difference between results of methods and results of simulation for example 21.

tx.	method	utilisation of bottleneck resource				
		0.4	0.5	0.6	0.7	0.8
T_1	M/G/1	-2.43	-5.73	-6.43	-10.72	-19.50
	max. ent.	-2.37	-5.66	-6.33	-10.61	-19.37
	decomp.	-2.41	-5.70	-6.38	-10.63	-19.36
	mean val.	9.54	10.15	14.74	16.40	14.20
T_2	M/G/1	14.18	17.33	22.71	26.43	32.67
	max. ent.	14.94	18.36	23.99	27.81	33.95
	decomp.	14.22	17.39	22.80	26.53	32.78
	mean val.	3.00	3.24	5.35	5.9	8.41
T_3	M/G/1	0.12	-0.93	0.37	0.81	1.30
	max. ent.	0.09	-0.99	0.27	0.64	1.04
	decomp.	0.11	-0.95	0.34	0.76	1.22
	mean val.	5.35	5.40	7.89	9.39	10.86

Table B-4 Percentage difference between results of methods and results of simulation for example 31.

Tables B-5 and B-6 present the results from the application of the heuristic rule to each of the examples discussed in Chapter 5. Columns 1 and 2 show the example id and the labelling produced by the rule. Columns 3-6 give the magnitude of the relative error for each value of the bottleneck utilisation (40% - 80%).

Id.	rule prediction	absolute relative error for given utilisation			
		below 5%	between 5% & 10%	between 10% & 15%	above 15%
01 T_1	A→M/M/1		0.4,0.5,0.6,0.7,0.8		
T_2	B,C→M/G/1	0.4,0.5,0.6,0.7,0.8			
02 T_1	A→M/G/1	0.4,0.5,0.6,0.7,0.8			
T_2	B,C→M/M/1	0.4,0.5,0.6,0.7,0.8			
03 T_1	A,C→M/M/1	0.4,0.5,0.6,0.7	0.8		
T_2	B→M/G/1	0.4,0.5,0.6,0.7,0.8			
04 T_1	A→M/M/1	0.4,0.7,0.8	0.5,0.6		
T_2	B,C→M/G/1	0.4,0.5,0.6,0.7,0.8			
05 T_1	A→M/M/1	0.4,0.8	0.5,0.6,0.7		
T_2	B,C→M/G/1	0.4,0.5,0.6,0.7,0.8			
06 T_1	A→M/M/1	0.4,0.5,0.8	0.6,0.7		
T_2	B,C→M/G/1	0.4,0.5,0.7	0.6,0.8		
07 T_1	A,B→M/M/1	0.4,0.5,0.7,0.8	0.6		
T_2	C→M/G/1	0.4,0.5,0.6,0.8	0.7		
08 T_1	A,B,C→M/G/1	0.4,0.5,0.6,0.7	0.8		
T_2		0.4,0.5,0.6	0.7	0.8	
09 T_1	A→M/M/1		0.4,0.5,0.8	0.6,0.7	
T_2	B,C→M/G/1	0.7,0.8	0.4,0.5,0.6		
21 T_1	A→M/M/1	0.8	0.4,0.5,0.6,0.7		
T_2	B→M/G/1	0.4,0.5,0.6	0.7	0.8	
T_3	C→M/G/1	0.6,0.7,0.8	0.4,0.5		

Table B-5 Results from heuristic rule

id.	rule prediction	absolute relative error for given utilisation			
		below 5%	between 5% & 10%	between 10% & 15%	above 15%
11 T	Disc0,1,2,3→M/G/1,PU→ M/G/1,SSU→M/G/1,NET →M/G/1	0.4,0.5,0.6	0.7,0.8		
12 T	Disc0,1,2,3→M/G/1,PU→ M/G/1,SSU→M/G/1,NET →M/G/1	0.4,0.5,0.6,0.7,0.8			
13 T	Disc0,1,2,3→M/G/1,PU→ M/G/1,SSU→M/G/1,NET →M/G/1	0.4,0.5,0.6,0.7,0.8			
14 T	Disc0,1,2,3→M/G/1,PU→ M/G/1,SSU→M/G/1,NET →M/G/1	0.4,0.5,0.6,0.7,0.8			
15 T	Disc0,1,2,3→M/G/1,PU→ M/G/1,SSU→M/G/1,NET →M/M/1	0.4,0.5,0.6	0.7	0.8	
16 T	Disc0,1,2,3→M/G/1,PU→ M/M/1,SSU→M/G/1,NET →M/G/1	0.4,0.5,0.6,0.7,0.8			
17 T	Disc0,1,2,3→M/G/1,PU→ M/M/1,SSU→M/G/1,NET →M/G/1	0.4	0.5,0.6	0.8	0.7 (-17.59%)
18 T	Disc0,1,2,3→M/G/1,PU→ M/G/1,SSU→M/G/1,NET →M/G/1	0.4,0.5,0.6,0.7,0.8			
19 T	Disc0,1,2,3→M/G/1,PU→ M/M/1,SSU→M/G/1,NET →M/M/1	0.4,0.5,0.7,0.8	0.6		
31 T_1 T_2 T_3	Disc0,1,3→M/G/1,PU-B→M/M/1,SSU-B→M/G/1,NET→M/M/1, PU-A→M/G/1,SSU-A→M/G/1,Disc2→M/M/1	0.4,0.5 0.4,0.5,0.6,0.7,0.8	0.4,0.5,0.8 0.6,0.7,0.8	0.6,0.7	

Table B-6 Results from heuristic rule

References

[1] A. Dan. "Performance analysis of data sharing environments", The MIT Press, 1992.

[2] A. Dan, P. Yu, "Performance analysis of coherency control policies through lock retention", in *Proc. of 1992 ACM SIGMOD Int. Conf. on Management of Data*, pp. 114-123, 1992.

[3] S. Zhou, M.H. Williams, H. Taylor, "Practical throughput estimation for parallel databases", *Software Engineering Journal*, vol.11, no.4, pp.255- 263, Jul 1996.

[4] S. Zhou, "Design of a DBMS performance estimator – STEADY", *Technical report PYTH.HWU.028*, Dept. of Comp. and El. Eng., Heriot-Watt Univ., Edinburgh, 1995.

[5] D. Dewitt, J. Gray, "Parallel database systems: the future of database processing or a passing fad", SIGMOD Record, vol.19, no.4, pp.104-112, Dec 1990.

[6] Informix Software Inc., "Informix-OnLine dynamic server administrator's guide", Informix Press, California, 1994.

[7] Oracle Corp., "Oracle7 parallel server concepts and administration, Release 7.3", 1996.

[8] C.K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland, W.G. Wilson, "DB2 parallel edition", *IBM Systems Journal*, vol.34, no.2, pp.292-322, 1995.

[9] C. Ballinger, "Teradata database design 101: a primer on Teradata physical database design and its advantages", *NCR Technical Journal*, http://192.127.252.174/journal/ball/index.htm, 1999.

[10] Compaq Computer Corporation, "Compaq NonStop SQL/MP", *Database Services Product Description*, http://www.tandem.com/prod_des/nssqlpd/nssqlpd.htm, 1999.

[11] T. Knoop, "Sybase SQL Server 11.0x technical overview", *Sybase Inc. white paper*, http://techinfo.sybase.com/css/techinfo.nsf/DocId/ID=8200-0996, March 1999.

[12] Microsoft Corp., "SQL Server 6.5 Evaluation Guide", *Technical white paper*, http://support.microsoft.com/support/sql/content/sql65/evalguid.asp, February, 1999.

[13] B. Bergsten, M. Couprie, P. Valduriez, "Prototyping DBS3, a shared-memory parallel database system", *Proc. of the 1st Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, USA, p.226-234, 4-6 Dec 1991.

[14] M. Zait, D. Florescu, P. Valduriez, "Benchmarking the DBS3 parallel query optimizer", *IEEE Parallel and Distributed Technology*, vol.4, no.2, pp.26-40, 1996.

[15] G. Graefe, "Parallelizing the Volcano database query processor", *Proc. 35th IEEE Computer Society Int. Conf.*, pp.490-493, 26 Feb - 02 Mar 1990.

[16] G. Graefe, S. Thakkar, "Tuning a parallel database algorithm on a shared-memory multiprocessor", *Software - Practice and Experience*, vol.22, no.7, pp.495-517, Jul 1992.

[17] W. Hong, M. Stonebraker, "Optimization of parallel query execution plans in XPRS", *Distributed and Parallel Databases*, vol.1, no.1, pp.9-32, Jan 1993.

[18] H. Lu, B. Ooi, K. Tan, "Query processing in parallel relational database systems", IEEE Computer Society Press, Los Alamitos, California, 1994.

[19] A. Shatdal, "Architectural considerations for parallel query evaluation algorithms", *PhD Dissertation*, University of Wisconsin-Madison, 1996.

[20] Sun Microsystems, "Sun Enterprise 10000 Server", *Technical white paper*, http://www.sun.com/servers/white-papers/E10000.pdf, September 1998.

[21] M. Stonebraker, "The case for shared-nothing", Database Engineering, vol. 9, no. 1, pp. 4-9, March 1986.

[22] D. Dewitt, G. Graefe, K. B. Kumar, R. H. Gerber, M. L. Heytens, M. Muralikrishna, "GAMMA - a high performance dataflow database machine", in *Proc. 12th Int. Conf. on Very Large Data Bases (VLDB '86)*, Kyoto, Japan, pp.228-237, 25-28 Aug 1986.

[23] D. Dewitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.I. Hsiao, R. Rasmussen, "Gamma database machine project", *IEEE Transactions on Knowledge and Data Engineering*, vol.2, no.1, pp.44-62, Mar 1990.

[24] H. Boral, "Parallelism in Bubba", in *Proc. Int. Symp. on Databases in Parallel and Distributed Systems*, Austin, TX, USA, pp.68-71, 5-7 Dec 1988.

[25] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, P. Valduriez, "Prototyping Bubba, a highly parallel database system", *IEEE Transactions on Knowledge and Data Engineering*, vol.2, no.1, pp.4-24, Mar 1990.

[26] P. Watson, P. Townsend, "The EDS parallel relational database system", in P. America (ed.), *Proc. of Workshop on Parallel Database Systems*, Noordwijk, the Netherlands, pp. 149-166, Sept. 1991.

[27] Informix Software Inc., "Informix-OnLine extended parallel server for loosely coupled cluster and massively parallel processing architectures", *Informix White Paper*, http://www.informix.com, 1998.

[28] C. J. White, "IBM System/390 and DB2 UDB: a high performance data warehouse server", *DB2 publications*, http://www.s390.ibm.com/ftp/marketing/position/ibmos390_version_2.pdf, May 1999.

[29] P. Valduriez, "Parallel database systems: the case for shared-something", in *Proc. 9th Int. Conf. on Data Engineering*, Vienna, Austria, pp.460-465, 19-23 Apr 1993.

[30] M. Garth, "Modelling parallel architectures", *Metron Technology white paper*, http://www.metron.co.uk/papers.htm#PARA, 1996.

[31] K. Rudin, "Running parallel", *DBMS Magazine*, issue 11, pp. 45-50, April 96.

[32] E.F. Codd, "A relational model of data for large shared data banks", in M. Stonebraker (ed.), *Readings in Database Systems*, 2nd Edition, pp.5-15, 1994.

[33] W. Hasan, "Optimisation of SQL queries for parallel machines", *PhD Dissertation*, Stanford University, Dec. 1995.

[34] D. Dewitt, J. Gray, "Parallel database systems. The future of high performance database systems", *Communications of the ACM*, vol.35, no.6, pp.85-98, Jun 1992.

[35] G. Graefe, "Encapsulation of parallelism in the volcano query processing system", *SIGMOD Record*, vol.19, no.2, pp.102-111, 1990.

[36] G. Graefe, D.L. Davison, "Encapsulation of parallelism and architecture-independence in extensible database query execution", *IEEE Transactions on Software Engineering*, vol.19, no.8, pp.749-763, Aug 1993.

[37] M.H. Williams, S. Zhou, "Data placement in parallel database systems", in M. Adbelguerfi, K.F. Wong (eds.), *Parallel Database Techniques*, IEEE Press, pp. 203-219, 1998.

[38] M. Mehta. D. Dewitt, "Data placement in shared-nothing parallel database systems", *VLDB Journal*, vol.6, no.1, pp.53-72, Feb 1997.

[39] D. Dewitt, R. Gerber, "Multiprocessor hashed-based join algorithms", in *Proc. 11th Int. Conf. Very Large Database Systems*, pp. 151-164, 1985.

[40] D. Dewitt et al, "Implementation techniques for main memory database systems", in *Proc. 1984 ACM SIGMOD Conf.*, pp. 1-8, 1984.

[41] H. Lu, K.L. Tan, M.C. Shan, "Hash-based join algorithms for multiprocessor computers with shared memory", in *Proc. 16th Int. Conf. Very Large Data Bases*, pp.198-209, 1990.

[42] D. A. Schneider, D. Dewitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment", *Proc. 1989 ACM Int. Conf. Management Data*, pp.110-121, 1989.

[43] B. Von Bultzingsloewen, "Optimising SQL queries for parallel execution", in *ACM SIGMOD Record*, vol. 18, no. 4, pp. 17-22, Dec. 1989.

[44] S. Ganguly, W. Hasan, R. Krishnamurthy, "Query optimisation for parallel execution". In *Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data*, San Diego, California, pp. 9-18, June 1992.

[45] J. Gray, "The benchmark handbook for database and transaction processing systems", 2nd edition, 1993.

[46] F. Raab, "Overview of the TPC benchmark C: a complex OLTP benchmark", in J. Gray (ed.), *The benchmark handbook for database and transaction processing systems*, 2nd edition, pp. 131-267, 1993.

[47] The Transaction Processing Council, http://www.tpc.org, 1999.

[48] C. Turbyfill, C. Orji, D. Bitton, "AS3AP: an ANSI SQL standard scaleable ad portable benchmark for relational database systems", in J. Gray (ed.), *The benchmark handbook for database and transaction processing systems*, 2nd edition, pp. 317-357, 1993.

[49] MERCURY Consortium, "MERCURY: Performance Management of Commercial Parallel Database Systems. Technical Annex for ESPRIT R&D Contract 20089", Sept. 1995.

[50] P. Watson, G. Catlow, "The architecture of the ICL Goldrush MegaServer", *ICL Systems Journal*, vol. 10, no. 2, http://www.icl.com/sjournal/v10i2/v10i2a1.html, November 1995.

[51] P. Watson, T. Robinson, "The hardware architecture of the ICL Goldrush MegaServer", *ICL Systems Journal*, vol. 10, no. 2, http://www.icl.com/sjournal/v10i2/v10i2a2.html, November 1995.

[52] P. Watson, G. Catlow, "The architecture of the ICL Goldrush MegaServer", In *Proc. of the 13th British National Conference on Databases (BNCOD 13)*, Manchester, U.K., pp. 250-262, 1995.

[53] S. Zhou, N. Tomov, "Some aspects of parallel servers on Goldrush", *MERCURY technical report* (*MERCURY.HWU.003*), Dep. Comp. and El. Eng., Heriot-Watt University, June 1996.

[54] J.A. Rolia, "Predicting the performance of software systems", *Technical report CSRI-260*, Computer Systems Research Institute, University of Toronto, Canada, January 1992.

[55] M. Molloy, "Fundamentals of performance modelling", Macmillan Publishing Company, 1989.

[56] M.K. Molloy, "Performance analysis using stochastic Petri nets", *IEEE Transactions on Computers*, vol. C-31, no. 9, pp. 913-917, Sept. 1982.

[57] M. Ajmone-Marsan, G. Balbo, G. Conte, "A class of generalised stochastic Petri nets for the performance evaluation of multi-processor systems", *ACM Trans. Comput. Systems*, vol. 2, no. 2, pp. 93-122, May 1984.

[58] L. Brunie, H. Kosch, W. Wohner, "From the modelling of parallel relational query processing to query optimization and simulation", *Parallel Processing Letters*, vol.8, no.1, pp.51-62, March 1998.

[59] J. Hillston, "A compositional approach for performance modelling", *PhD Thesis*, University of Edinburgh, UK, 1994.

[60] C. Pua, "Process algebra approach to parallel DBMS performance modelling", *PhD Dissertation*, Heriot-Watt University, 1999.

[61] D. Redfern, C. Campbell, "The Matlab 5 handbook", Springer, 1998.

[62] E. Gelenbe, G. Pujolle, "Introduction to queueing networks", John Wiley & Sons Ltd., 1987.

[63] G. Bolch, S. Greiner, H. de Meer, K.S. Trivedi, "Queueing networks and Markov chains: modelling and performance evaluation with computer science applications", John Wiley & Sons Inc., 1998.

[64] L. Kleinrock, "Queueing systems Vol. 1 – Theory", John Wiley and Sons, 1975.

[65] F. Baskett, K. Chandy, R. Muntz, F. Palacios, "Open, closed and mixed networks of queues with different classes of customers", *Journal of the ACM*, vol. 22, no. 2, pp. 248-260, April 1975.

[66] J. Jackson, "Networks of waiting lines", *Operations Research*, vol. 5, no. 4, pp.518-521, 1957.

[67] W. Gordon, G. Newell, "Closed queueing systems with exponential servers", *Operations Research*, vol. 15, no. 2, pp. 254-265, April 1967.

[68] A. Duda, T. Czachorski, "Performance evaluation of fork and join synchronisation primitives", *Acta Informatica*, vol. 24, pp. 525-553, 1987

[69] J.A. Rolia, K.C. Sevcik, "The method of layers", *IEEE Trans. on Software Engineering*, vol. 21, no. 8, pp. 689-700, August 1995.

[70] G. Franks, A. Hubbard, S. Majumdar, D. Petriu, J. Rolia, M. Woodside, "A toolset for performance engineering and software design of client-server systems", *Performance Evaluation*, vol. 24, no. 1-2, pp. 117-135, November 1995.

[71] A. Bhide, "An analysis of three transaction processing architectures", in *Proc. of 14th VLDB Conf.*, pp. 339-350, 1988.

[72] E. Mohamed, H. El-Rewini, H. Abdel-Wahab, A. Helal, "Parallel database architectures: a comparison study", *Informatica*, vol. 22, pp.195-205, 1998.

[73] K. Hua, C. Lee, "Handling data skew in multiprocessor database computers using partition tuning", in *Proc. of 17th Int. Conf. On Very Large Data Bases*, pp. 525-535, 1991.

[74] H. Lu, K. Tan, "Dynamic and load-balanced task-oriented database query processing in parallel systems", in *Proc. of 3rd Int. Conf. Extending Database Technology*, pp. 357-372, 1992.

[75] M. S. Lakshmi, P. S. Yu, "Effectiveness of parallel joins", *IEEE Trans. Knowledge and Data Engineering*, vol. 2, no. 4, pp. 410-424, Dec. 1990.

[76] E. Omiecinski, "Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor", in *Proc. 17th Int. Conf. Very Large Data Bases*, pp. 375-385, 1991.

[77] C. Walton, A. Dale, R. Jenevein, "A taxonomy and performance model of data skew effects in parallel joins", In *Proc. of the 17th Int. Conf. on Very Large Data Bases*, Barcelona, Spain, pp.537-548, September 1991.

[78] M. Garofalakis, Y. E. Ioannidis, "Multi-dimensional resource scheduling for parallel queries", In *Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data*, Montreal, Canada, pp.365-376, June 1996.

[79] M. S. Chen, P. S. Yu, K.L. Wu, "Optimization of parallel execution for multi-join queries", *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 3, pp. 416-428, Jun 1996.

[80] K. L. Tan, H. Lu, "Scheduling multiple queries in symmetric multiprocessors", *Information Sciences*, vol.95, no.1-2, pp.125-153, Nov 1996.

[81] H. Hsiao, M. Chen, P. Yu, "On parallel execution of multiple pipelined hash joins", In *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, Minnesota, pp.185-196, May 1994.

[82] J. P. Richardson, H. Lu, K. Mikkilineni, "Design and evaluation of parallel pipelined join algorithms", in *Proc. of ACM SIGMOD*, pp. 399-409, 1987.

[83] K. Sevcik, "Data base system performance prediction using an analytical model", In *Proc. of the 7th Int. Conf. on Very Large Data Bases*, Cannes, France, pp. 182-197, September, 1981.

[84] S. Salza, M. Renzetti, "Performance modelling of parallel database systems", *Informatica*, vol.22, pp.127-139, 1998.

[85] S. Salza, R. Tomasso, "A modelling tool for the performance analysis of relational database applications", In *Proc. 6th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pp.323-337, 1992.

[86] M. Spiliopoulou, J. C. Freytag, "Modelling the dynamic evolution of system workload during pipelined query execution", *Technical Report ISS-20*, Institut für Wirtschaftsinformatik, Humboldt-Universität zu Berlin, Germany, 1995.

[87] M. Spiliopoulou, J. C. Freytag, "Modelling resource utilisation in pipelined query execution", in *Proc. of 2nd Int. Euro-Par Conf.*, Lyon, France, pp. 872-880, 26-29 Aug, 1996.

[88] M. Spiliopoulou, M. Hatzopoulos, Y. Cotronis, "Parallel optimization of large join queries with set operators and aggregates in a parallel environment supporting pipeline", *IEEE Trans. On Knowledge and Data Engineering*, vol. 8, no.3, pp.429-445, June 1996.

[89] R. Eberhard, IBM Corp. "DB2 Estimator for Windows", http://www.software.ibm.com/data/db2/os390/estimate, 1999.

[90] J. Boulos, D. Boudigue, "An application of SMART2: a tool for performance evaluation of relational database programs", in *Joint proc. of 8th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation and 8th GI/ITG Conf. on Measuring, Modelling and Evaluating Computing and Communication Systems*, Heidelberg, Germany, pp.11-25, 20-22 Sept. 1995.

[91] J.L. Anciano, N.N. Savino, J.A. Corbacho, R. Puigjaner, "Extending SMART2 to predict the behaviour of PL/SQL-based applications", in *Proc. of 10th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation (Tools'98)*, Palma de Mallorca, Spain, pp.292-305, 14-18 Sept. 1998.

[92] Dell Computer Corp., "Oracle System Sizer", http://www.dell.com/products/poweredge/partners/db/oracle/wp-sizer.htm, 1998.

[93] M. Garth, "Capacity planning for parallel IT systems", *The Computer Bulletin*, vol. 8, no. 5, pp. 16-18, November 1996.

[94] T. Foxon, M. Garth, P. Harrison, "Capacity planning in client-server systems", *Journal of Distributed Systems Engineering*, vol. 3, pp. 32-38, 1996.

[95] T. Foxon, "Performance Analysis of an Oracle-based interactive UNIX system – a case study", *Metron Technology white paper*, http://www.metron.co.uk/papers.htm, 1994.

[96] Platinum Technology, "Proactive performance engineering", *Platinum Technology white paper*, http://www.softool.com/products/ppewhite.htm, 1999.

[97] G. Sigalov, B. Zibitsker, "Performance evaluation of database computers with high level of parallel processing", In *Proc.of the 19th Int. Conf. for the Management and Performance Evaluation of Enterprise Computing Systems*, San Diego, California, pp. 1100-1109, December 1993.

[98] BEZ Systems Inc., "BEZPlus for NCR Teradata and Oracle environments on MPP machines", http://www.bez.com/software.htm, 1999.

[99] SES Inc., "Solutions for information systems performance", http://www.ses.com/Solution/IS.html, 1999.

[100] M.H. Williams, E. Dempster, N. Tomov, C. Pua, H. Taylor, A. Burger, J. Lu, P. Broughton, "An analytical tool for predicting the performance of parallel relational databases", *Concurrency: Practice and Experience*, to appear, 1999.

[101] Ingres Corporation, "Relational system – Ingres database administrator's guide", Release 6.4, December 1991.

[102] K. Hua, C. Lee, H. Young, "An efficient load balancing strategy for shared-nothing database systems", in *Proc. of Database and Expert Systems Applications 92*, Valencia, Spain, pp.469-474, 1992.

[103] E. Gelenbe, I. Mitrani, "Analysis and synthesis of computer systems", Academic Press Inc., London, 1980.

[104] D. Kouvatsos, H. Georgatsos, N. Tabet-Aouel, "A universal maximum entropy algorithm for general multiple class open networks with mixed service disciplines", *Technical report DDK/PHG-1, Computing Systems Modelling Research Group, Bradford University, England*, 1988.

[105] D. Kouvatsos, (1985) "Maximum entropy methods for general queueing networks", In D. Potier (ed), *Modelling Techniques and Tools for Performance Analysis*, Elsevier Science Publishers B. V. (North-Holland), pp. 589-608, 1985.

[106] R. Walstra, "Nonexponential networks of queues: a maximum entropy analysis", *ACM SIGMETRICS Performance Evaluation Review*, vol.13, no.2, pp.27-37, 1985.

[107] P. Kuehn, "Approximate analysis of general queuing networks by decomposition", *IEEE Transactions on Communications*, vol.27, no.1, pp.113-126, 1979.

[108] G. Pujolle, W. Ai, "A solution for multiserver and multiclass open queueing networks", *INFOR*, vol.24, no.3, pp.221-230, 1986.

[109] W. Whitt, "The queueing network analyser", *The Bell System Technical Journal*, vol.62, pp.2779-2815, 1983.

[110] W. Whitt, "Performance of the queueing network analyser", *The Bell System Technical Journal*, vol.62, pp. 2817-2843, 1983.

[111] J.M. Harrison, V. Nguyen, "The QNET method for two-moment analysis of open queueing networks", *Queueing Systems: Theory & Applications*, vol.6, pp.1-32, 1990.

[112] Mesquite Software, Inc. "CSIM18 simulation engine user guide", 1998.

[113] E. Dempster, "Initial calibration queries", *Technical report MERCURY.HWU.029*, Dept. of Comp. and El. Eng, Heriot-Watt Univ., Edinburgh, 1998.

[114] E. Dempster, "Initial calibration queries", *Technical report MERCURY.HWU.040*, Dept. of Comp. and El. Eng, Heriot-Watt Univ., Edinburgh, 1998.

[115] C. Mohan, I. Narang, "Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disc transaction environment", in *Proc. of 17th Int. Conf. Very Large Data Bases*, Barcelona, Spain, pp. 193-207, Sep. 1991.

[116] K. Wilkinson, M. Neimat, "Maintaining consistency of client-cached data", in *Proc. of the 16th Int. Conf. Very Large Data Bases*, Brisbane, Australia, pp. 122-133, August 1990.

[117] Y. Wang, L.A. Rowe, "Cache consistency and concurrency control in a client/server DBMS architecture", in *Proc. ACM SIGMOD Conf*, Boulder, pp. 367-376, 1991.

[118] A. Dan, P. Yu, "Performance analysis of buffer coherency policies in a multisystem data sharing environment", *IEEE Trans. On Parallel and Distributed Systems*, vol. 4, no. 3, pp. 289-305, March 1993.

[119] D. Dias, B. Iyer, P. Yu, "Inegrated concurrency-coherency controls for multisystem data sharing", *IEEE Trans. Soft. Eng.*, vol. 15, no. 4, pp.437-448, 1989.

[120] E. Rahm, "Concurrency and coherency control in database sharing systems", *Technical report ZRI 3/91*, Dept. of Computer Science, University of Kaiserslautern, Germany, revised August 1992.

[121] A. Thomasian, "Concurrency control: methods, performance, and analysis", *ACM Computing Surveys*, vol.30, no.1, pp.70-119, March 1998.

[122] P. S. Yu, D.M. Dias, S.S. Lavenberg, "On the analytical modeling of database concurrency control", *Journal of the ACM*, vol. 40, no. 4, pp. 831-872, Sept. 1993.

[123] D. Messham, "Trimetra Xtraserver", *ICL Systems Journal*, vol. 13, no. 1, http://www.icl.com/sjournal/v13i1/V13i1all.html#Xtra, Autumn 1998.

[124] P. Broughton, "Final report and deliverable for MERCURY project", *ESPRIT R&D Contract 20089*, part 2, pp. 3, July 1999.